

Effective Software Development Series   
No Starch Press, Inc. [nostarch.com](http://nostarch.com)

# *Effective* PYTHON

90 Specific Ways to Write Better Python

SECOND EDITION



Brett Skottman

# Contents

Cover Page

About This eBook

Half Title Page

Title Page

Copyright Page

Dedication Page

Contents at a Glance

Contents

Preface

Acknowledgments

About the Author

## 1. Pythonic Thinking

Item 1: Know Which Version of Python You're Using

Item 2: Follow the PEP 8 Style Guide

Item 3: Know the Differences Between `bytes` and `str`

Item 4: Prefer Interpolated F-Strings Over C-style Format Strings  
and `str.format`

Item 5: Write Helper Functions Instead of Complex Expressions

Item 6: Prefer Multiple Assignment Unpacking Over Indexing

Item 7: Prefer `enumerate` Over `range`

Item 8: Use `zip` to Process Iterators in Parallel

Item 9: Avoid `else` Blocks After `for` and `while` Loops

Item 10: Prevent Repetition with Assignment Expressions

## 2. Lists and Dictionaries



Item 11: Know How to Slice Sequences

Item 12: Avoid Striding and Slicing in a Single Expression

Item 13: Prefer Catch-All Unpacking Over Slicing

Item 14: Sort by Complex Criteria Using the `key` Parameter

Item 15: Be Cautious When Relying on dict Insertion Ordering

Item 16: Prefer `get` Over `in` and `keyError` to Handle Missing Dictionary Keys

Item 17: Prefer `defaultdict` Over `setdefault` to Handle Missing Items in Internal State

Item 18: Know How to Construct Key-Dependent Default Values with `__missing__`

### 3. Functions

Item 19: Never Unpack More Than Three Variables When Functions Return Multiple Values

Item 20: Prefer Raising Exceptions to Returning `None`

Item 21: Know How Closures Interact with Variable Scope

Item 22: Reduce Visual Noise with Variable Positional Arguments

Item 23: Provide Optional Behavior with Keyword Arguments

Item 24: Use `None` and Docstrings to Specify Dynamic Default Arguments

Item 25: Enforce Clarity with Keyword-Only and Positional-Only Arguments

Item 26: Define Function Decorators with `functools.wraps`

### 4. Comprehensions and Generators

Item 27: Use Comprehensions Instead of `map` and `filter`

Item 28: Avoid More Than Two Control Subexpressions in Comprehensions

Item 29: Avoid Repeated Work in Comprehensions by Using Assignment Expressions

Item 30: Consider Generators Instead of Returning Lists

Item 31: Be Defensive When Iterating Over Arguments

Item 32: Consider Generator Expressions for Large List Comprehensions

Item 33: Compose Multiple Generators with `yield from`

Item 34: Avoid Injecting Data into Generators with `send`

Item 35: Avoid Causing State Transitions in Generators with `throw`

Item 36: Consider `itertools` for Working with Iterators and Generators

## 5. Classes and Interfaces

Item 37: Compose Classes Instead of Nesting Many Levels of Built-in Types

Item 38: Accept Functions Instead of Classes for Simple Interfaces

Item 39: Use `@classmethod` Polymorphism to Construct Objects Generically

Item 40: Initialize Parent Classes with `super`

Item 41: Consider Composing Functionality with Mix-in Classes

Item 42: Prefer Public Attributes Over Private Ones

Item 43: Inherit from `collections.abc` for Custom Container Types

## 6. Metaclasses and Attributes

Item 44: Use Plain Attributes Instead of Setter and Getter Methods

Item 45: Consider `@property` Instead of Refactoring Attributes

Item 46: Use Descriptors for Reusable `@property` Methods

Item 47: Use `__getattr__`, `__getattribute__`, and `__setattr__` for Lazy Attributes

Item 48: Validate Subclasses with `__init_subclass__`

Item 49: Register Class Existence with `__init_subclass__`

Item 50: Annotate Class Attributes with `__set_name__`

Item 51: Prefer Class Decorators Over Metaclasses for Composable Class Extensions

## 7. Concurrency and Parallelism

Item 52: Use `subprocess` to Manage Child Processes

Item 53: Use Threads for Blocking I/O, Avoid for Parallelism

Item 54: Use `Lock` to Prevent Data Races in Threads

Item 55: Use `queue` to Coordinate Work Between Threads

Item 56: Know How to Recognize When Concurrency Is Necessary

Item 57: Avoid Creating New Thread Instances for On-demand Fan-out

Item 58: Understand How Using `queue` for Concurrency Requires Refactoring

Item 59: Consider `ThreadPoolExecutor` When Threads Are Necessary for Concurrency

Item 60: Achieve Highly Concurrent I/O with Coroutines

Item 61: Know How to Port Threaded I/O to `asyncio`

Item 62: Mix Threads and Coroutines to Ease the Transition to `asyncio`

Item 63: Avoid Blocking the `asyncio` Event Loop to Maximize Responsiveness

Item 64: Consider `concurrent.futures` for True Parallelism

## 8. Robustness and Performance

Item 65: Take Advantage of Each Block in  
`try/except/else/finally`

Item 66: Consider `contextlib` and `with` Statements for Reusable  
`try/finally` Behavior

Item 67: Use `datetime` Instead of `time` for Local Clocks

Item 68: Make `pickle` Reliable with `copyreg`

Item 69: Use `decimal` When Precision Is Paramount

Item 70: Profile Before Optimizing

Item 71: Prefer `deque` for Producer–Consumer Queues

Item 72: Consider Searching Sorted Sequences with `bisect`

Item 73: Know How to Use `heapq` for Priority Queues

Item 74: Consider `memoryview` and `bytearray` for Zero-Copy  
Interactions with `bytes`

## 9. Testing and Debugging

Item 75: Use `repr` Strings for Debugging Output

Item 76: Verify Related Behaviors in `TestCase` Subclasses

Item 77: Isolate Tests from Each Other with `setUp`, `tearDown`,  
`setUpModule`, and `tearDownModule`

Item 78: Use Mocks to Test Code with Complex Dependencies

Item 79: Encapsulate Dependencies to Facilitate Mocking and  
Testing

Item 80: Consider Interactive Debugging with `pdb`

Item 81: Use `tracemalloc` to Understand Memory Usage and  
Leaks

## 10. Collaboration

Item 82: Know Where to Find Community-Built Modules

Item 83: Use Virtual Environments for Isolated and Reproducible  
Dependencies

Item 84: Write Docstrings for Every Function, Class, and Module

Item 85: Use Packages to Organize Modules and Provide Stable APIs

Item 86: Consider Module-Scoped Code to Configure Deployment Environments

Item 87: Define a Root Exception to Insulate Callers from APIs

Item 88: Know How to Break Circular Dependencies

Item 89: Consider warnings to Refactor and Migrate Usage

Item 90: Consider Static Analysis via typing to Obviate Bugs

Index

Code Snippets

i

ii

iii

iv

v

vi

vii

viii

ix

x

xi

xii

xiii

xiv

xv

xvi

xvii

xviii

xix

xx

xxi

xxii

xxiii

xxiv

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73



74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100

101

102

103

104

105

106

107

108

109

110

111

112

113

114

115

116

117

118

119

120

121

122

123

124

125

126

127

128

129

130

131

132

133

134

135

136

137

138

139

140

141

142

143

144

145

146

147

148

149

150

151

152

153

154

155

156

157

158

159

160

161

162

163

164

165

166

167

168

169

170

171

172

173

174

175

176

177

178

179

180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

205

206

207

208

209

210

211

212

213

214

215

216

217

218

219

220

221

222

223

224

225

226

227

228

229

230

231

232

233

234

235

236

237

238

239

240

241

242

243

244

245

246

247

248

249

250

251

252

253

254

255

256

257

258

259

260

261

262

263

264

265

266

267

268

269

270

271

272

273

274

275

276

277

278

279

280

281

282

283

284

285

286

287

288

289



290

291

292

293

294

295

296

297

298

299

300

301

302

303

304

305

306

307

308

309

310

311

312

313

314

315

316

317

318

319

320

321

322

323

324

325

326

327

328

329

330

331

332

333

334

335

336

337

338

339

340

341

342

343

344

345

346

347

348

349

350

351

352

353

354

355

356

357

358

359

360

361

362

363

364

365

366

367

368

369

370

371

372

373

374

375

376

377

378

379

380

381

382

383

384

385

386

387

388

389

390

391

392

393

394

395

396

397

398

399

400

401

402

403

404

405

406

407

408

409

410

411

412

413

414

415

416

417

418

419

420

421

422

423

424

425

426

427

428

429

430

431

432

433

434

435

436

437

438

439

440

441

442

443

444

445

446

447

448

449

450

451

452

453

454

455

456

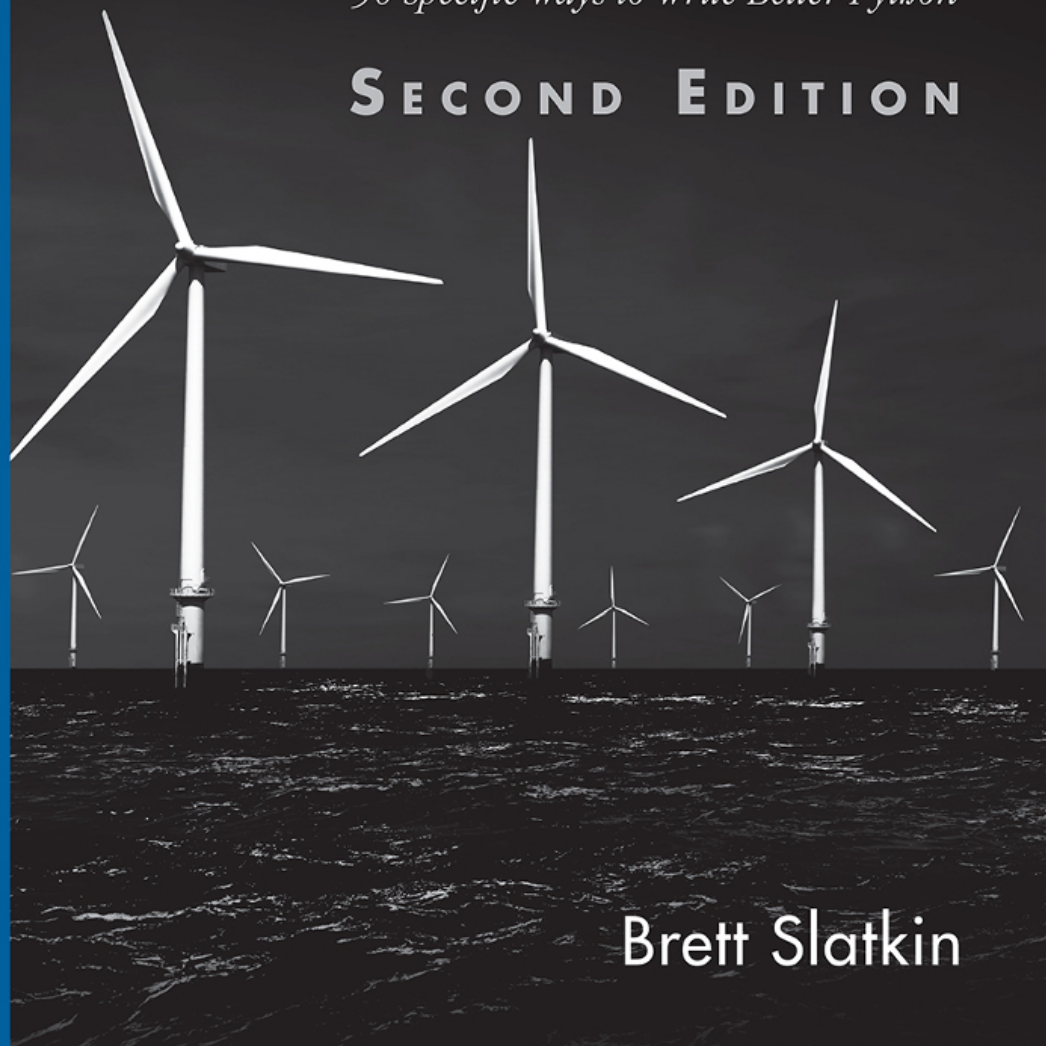
*Effective* SOFTWARE DEVELOPMENT SERIES  
Scott Meyers, Consulting Editor



# *Effective* PYTHON

*90 Specific Ways to Write Better Python*

SECOND EDITION



Brett Slatkin



## About This eBook

ePUB is an open, industry-standard format for eBooks. However, support of ePUB and its many features varies across reading devices and applications. Use your device or app settings to customize the presentation to your liking. Settings that you can customize often include font, font size, single or double column, landscape or portrait mode, and figures that you can click or tap to enlarge. For additional information about the settings and features on your reading device or app, visit the device manufacturer's Web site.

Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the eBook in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a "Click here to view code image" link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.

## **Praise for *Effective Python***

---

“I have been recommending this book enthusiastically since the first edition appeared in 2015. This new edition, updated and expanded for Python 3, is a treasure trove of practical Python programming wisdom that can benefit programmers of all experience levels.”

—Wes McKinney, *Creator of Python Pandas project, Director of Ursa Labs*

“If you’re coming from another language, this is your definitive guide to taking full advantage of the unique features Python has to offer. I’ve been working with Python for nearly twenty years and I still learned a bunch of useful tricks, especially around newer features introduced by Python 3. *Effective Python* is crammed with actionable advice, and really helps define what our community means when they talk about Pythonic code.”

—Simon Willison, *Co-creator of Django*

“I’ve been programming in Python for years and thought I knew it pretty well. Thanks to this treasure trove of tips and techniques, I’ve discovered many ways to improve my Python code to make it faster (e.g., using `bisect` to search sorted lists), easier to read (e.g., enforcing keyword-only arguments), less prone to error (e.g., unpacking with starred expressions), and more Pythonic (e.g., using `zip` to iterate over lists in parallel). Plus, the second edition is a great way to quickly get up to speed on Python 3 features, such as the walrus operator, f-strings, and the typing module.”

—Pamela Fox, *Creator of Khan Academy programming courses*

“Now that Python 3 has finally become the standard version of Python, it’s already gone through eight minor releases and a lot of new features have been added throughout. Brett Slatkin returns with a second edition of *Effective Python* with a huge new list of Python idioms and straightforward recommendations, catching up with everything that’s introduced in version 3 all the way through 3.8 that we’ll all want to use as we finally leave Python 2 behind. Early sections lay out an enormous list of tips regarding new Python 3 syntaxes and concepts like string and byte objects, f-strings,

assignment expressions (and their special nickname you might not know), and catch-all unpacking of tuples. Later sections take on bigger subjects, all of which are packed with things I either didn't know or which I'm always trying to teach to others, including 'Metaclasses and Attributes' (good advice includes 'Prefer Class Decorators over Metaclasses' and also introduces a new magic method '`__init_subclass__()`' I wasn't familiar with), 'Concurrency' (favorite advice: 'Use Threads for Blocking I/O, but not Parallelism,' but it also covers asyncio and coroutines correctly) and 'Robustness and Performance' (advice given: 'Profile before Optimizing'). It's a joy to go through each section as everything I read is terrific best practice information smartly stated, and I'm considering quoting from this book in the future as it has such great advice all throughout. This is the definite winner for the 'if you only read one Python book this year...' contest.

—Mike Bayer, *Creator of SQLAlchemy*

"This is a great book for both novice and experienced programmers. The code examples and explanations are well thought out and explained concisely and thoroughly. The second edition updates the advice for Python 3, and it's fantastic! I've been using Python for almost 20 years, and I learned something new every few pages. The advice given in this book will serve anyone well."

—Titus Brown, *Associate Professor at UC Davis*

"Once again, Brett Slatkin has managed to condense a wide range of solid practices from the community into a single volume. From exotic topics like metaclasses and concurrency to crucial basics like robustness, testing, and collaboration, the updated *Effective Python* makes a consensus view of what's 'Pythonic' available to a wide audience."

—Brandon Rhodes, *Author of python-patterns.guide*

# **Effective Python**

---

**Second Edition**

# **Effective Python**

---

**90 SPECIFIC WAYS TO WRITE BETTER PYTHON**

**Second Edition**

**Brett Slatkin**

◆◆ Addison-Wesley

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at [corpsales@pearsoned.com](mailto:corpsales@pearsoned.com) or (800) 382-3419.

For government sales inquiries, please contact [governmentsales@pearsoned.com](mailto:governmentsales@pearsoned.com).

For questions about sales outside the U.S., please contact [intlcs@pearson.com](mailto:intlcs@pearson.com).

Visit us on the Web: [informit.com/aw](http://informit.com/aw)

Library of Congress Control Number: On file

Copyright © 2020 Pearson Education, Inc.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit [www.pearson.com/permissions/](http://www.pearson.com/permissions/).

ISBN-13: 978-0-13-485398-7

ISBN-10: 0-13-485398-9

ScoutAutomatedPrintCode

**Editor-in-Chief**

Mark L. Taub

**Executive Editor**

Deborah Williams

**Development Editor**

Chris Zahn

**Managing Editor**

Sandra Schroeder

**Senior Project Editor**

Lori Lyons

**Production Manager**

Aswini Kumar/codeMantra

**Copy Editor**

Catherine D. Wilson

**Indexer**

Cheryl Lenser

**Proofreader**

Gill Editorial Services

**Cover Designer**

Chuti Prasertsith

**Compositor**

codeMantra

*To our family*



# **Contents at a Glance**

**Preface**

**Acknowledgments**

**About the Author**

**Chapter 1: Pythonic Thinking**

**Chapter 2: Lists and Dictionaries**

**Chapter 3: Functions**

**Chapter 4: Comprehensions and Generators**

**Chapter 5: Classes and Interfaces**

**Chapter 6: Metaclasses and Attributes**

**Chapter 7: Concurrency and Parallelism**

**Chapter 8: Robustness and Performance**

**Chapter 9: Testing and Debugging**

**Chapter 10: Collaboration**

**Index**

# Contents

## Preface

## Acknowledgments

## About the Author

## Chapter 1 Pythonic Thinking

- Item 1: Know Which Version of Python You're Using
- Item 2: Follow the PEP 8 Style Guide
- Item 3: Know the Differences Between `bytes` and `str`
- Item 4: Prefer Interpolated F-Strings Over C-style Format Strings and `str.format`
- Item 5: Write Helper Functions Instead of Complex Expressions
- Item 6: Prefer Multiple Assignment Unpacking Over Indexing
- Item 7: Prefer `enumerate` Over `range`
- Item 8: Use `zip` to Process Iterators in Parallel
- Item 9: Avoid `else` Blocks After `for` and `while` Loops
- Item 10: Prevent Repetition with Assignment Expressions

## Chapter 2 Lists and Dictionaries

- Item 11: Know How to Slice Sequences
- Item 12: Avoid Striding and Slicing in a Single Expression
- Item 13: Prefer Catch-All Unpacking Over Slicing
- Item 14: Sort by Complex Criteria Using the `key` Parameter
- Item 15: Be Cautious When Relying on `dict` Insertion Ordering
- Item 16: Prefer `get` Over `in` and `keyError` to Handle Missing Dictionary Keys

- Item 17: Prefer `defaultdict` Over `setdefault` to Handle Missing Items in Internal State
- Item 18: Know How to Construct Key-Dependent Default Values with `__missing__`

### Chapter 3 Functions

- Item 19: Never Unpack More Than Three Variables When Functions Return Multiple Values
- Item 20: Prefer Raising Exceptions to Returning `None`
- Item 21: Know How Closures Interact with Variable Scope
- Item 22: Reduce Visual Noise with Variable Positional Arguments
- Item 23: Provide Optional Behavior with Keyword Arguments
- Item 24: Use `None` and Docstrings to Specify Dynamic Default Arguments
- Item 25: Enforce Clarity with Keyword-Only and Positional-Only Arguments
- Item 26: Define Function Decorators with `functools.wraps`

### Chapter 4 Comprehensions and Generators

- Item 27: Use Comprehensions Instead of `map` and `filter`
- Item 28: Avoid More Than Two Control Subexpressions in Comprehensions
- Item 29: Avoid Repeated Work in Comprehensions by Using Assignment Expressions
- Item 30: Consider Generators Instead of Returning Lists
- Item 31: Be Defensive When Iterating Over Arguments
- Item 32: Consider Generator Expressions for Large List Comprehensions
- Item 33: Compose Multiple Generators with `yield from`
- Item 34: Avoid Injecting Data into Generators with `send`

Item 35: Avoid Causing State Transitions in Generators with `throw`

Item 36: Consider `itertools` for Working with Iterators and Generators

## Chapter 5 Classes and Interfaces

Item 37: Compose Classes Instead of Nesting Many Levels of Built-in Types

Item 38: Accept Functions Instead of Classes for Simple Interfaces

Item 39: Use `@classmethod` Polymorphism to Construct Objects Generically

Item 40: Initialize Parent Classes with `super`

Item 41: Consider Composing Functionality with Mix-in Classes

Item 42: Prefer Public Attributes Over Private Ones

Item 43: Inherit from `collections.abc` for Custom Container Types

## Chapter 6 Metaclasses and Attributes

Item 44: Use Plain Attributes Instead of Setter and Getter Methods

Item 45: Consider `@property` Instead of Refactoring Attributes

Item 46: Use Descriptors for Reusable `@property` Methods

Item 47: Use `__getattr__`, `__getattribute__`, and `__setattr__` for Lazy Attributes

Item 48: Validate Subclasses with `__init_subclass__`

Item 49: Register Class Existence with `__init_subclass__`

Item 50: Annotate Class Attributes with `__set_name__`

Item 51: Prefer Class Decorators Over Metaclasses for Composable Class Extensions

## Chapter 7 Concurrency and Parallelism

- Item 52: Use `subprocess` to Manage Child Processes
- Item 53: Use Threads for Blocking I/O, Avoid for Parallelism
- Item 54: Use `Lock` to Prevent Data Races in Threads
- Item 55: Use `queue` to Coordinate Work Between Threads
- Item 56: Know How to Recognize When Concurrency Is Necessary
- Item 57: Avoid Creating New Thread Instances for On-demand Fan-out
- Item 58: Understand How Using `queue` for Concurrency Requires Refactoring
- Item 59: Consider `ThreadPoolExecutor` When Threads Are Necessary for Concurrency
- Item 60: Achieve Highly Concurrent I/O with Coroutines
- Item 61: Know How to Port Threaded I/O to `asyncio`
- Item 62: Mix Threads and Coroutines to Ease the Transition to `asyncio`
- Item 63: Avoid Blocking the `asyncio` Event Loop to Maximize Responsiveness
- Item 64: Consider `concurrent.futures` for True Parallelism

## **Chapter 8 Robustness and Performance**

- Item 65: Take Advantage of Each Block in `try/except/else/finally`
- Item 66: Consider `contextlib` and `with` Statements for Reusable `try/finally` Behavior
- Item 67: Use `datetime` Instead of `time` for Local Clocks
- Item 68: Make `pickle` Reliable with `copyreg`
- Item 69: Use `decimal` When Precision Is Paramount
- Item 70: Profile Before Optimizing
- Item 71: Prefer `deque` for Producer–Consumer Queues
- Item 72: Consider Searching Sorted Sequences with `bisect`

- Item 73: Know How to Use `heapq` for Priority Queues
- Item 74: Consider `memoryview` and `bytearray` for Zero-Copy Interactions with bytes

## Chapter 9 Testing and Debugging

- Item 75: Use `repr` Strings for Debugging Output
- Item 76: Verify Related Behaviors in `TestCase` Subclasses
- Item 77: Isolate Tests from Each Other with `setUp`, `tearDown`, `setUpModule`, and `tearDownModule`
- Item 78: Use Mocks to Test Code with Complex Dependencies
- Item 79: Encapsulate Dependencies to Facilitate Mocking and Testing
- Item 80: Consider Interactive Debugging with `pdb`
- Item 81: Use `tracemalloc` to Understand Memory Usage and Leaks

## Chapter 10 Collaboration

- Item 82: Know Where to Find Community-Built Modules
- Item 83: Use Virtual Environments for Isolated and Reproducible Dependencies
- Item 84: Write Docstrings for Every Function, Class, and Module
- Item 85: Use Packages to Organize Modules and Provide Stable APIs
- Item 86: Consider Module-Scoped Code to Configure Deployment Environments
- Item 87: Define a Root `Exception` to Insulate Callers from APIs
- Item 88: Know How to Break Circular Dependencies
- Item 89: Consider warnings to Refactor and Migrate Usage
- Item 90: Consider Static Analysis via `typing` to Obviate Bugs

## Index

# Preface

The Python programming language has unique strengths and charms that can be hard to grasp. Many programmers familiar with other languages often approach Python from a limited mindset instead of embracing its full expressivity. Some programmers go too far in the other direction, overusing Python features that can cause big problems later.

This book provides insight into the *Pythonic* way of writing programs: the best way to use Python. It builds on a fundamental understanding of the language that I assume you already have. Novice programmers will learn the best practices of Python’s capabilities. Experienced programmers will learn how to embrace the strangeness of a new tool with confidence.

My goal is to prepare you to make a big impact with Python.

## What This Book Covers

Each chapter in this book contains a broad but related set of items. Feel free to jump between items and follow your interest. Each item contains concise and specific guidance explaining how you can write Python programs more effectively. Items include advice on what to do, what to avoid, how to strike the right balance, and why this is the best choice. Items reference each other to make it easier to fill in the gaps as you read.

This second edition of this book is focused exclusively on Python 3 (see [Item 1: “Know Which Version of Python You’re Using”](#)), up to and including version 3.8. Most of the original items from the first edition have been revised and included, but many have undergone substantial updates. For some items, my advice has completely changed between the two editions of the book due to best practices evolving as Python has matured. If you’re still primarily using Python 2, despite its end-of-life on January 1st, 2020, the previous edition of the book may be more useful to you.

Python takes a “batteries included” approach to its standard library, in comparison to many other languages that ship with a small number of

common packages and require you to look elsewhere for important functionality. Many of these built-in packages are so closely intertwined with idiomatic Python that they may as well be part of the language specification. The full set of standard modules is too large to cover in this book, but I've included the ones that I feel are critical to be aware of and use.

## **Chapter 1: Pythonic Thinking**

The Python community has come to use the adjective *Pythonic* to describe code that follows a particular style. The idioms of Python have emerged over time through experience using the language and working with others. This chapter covers the best way to do the most common things in Python.

## **Chapter 2: Lists and Dictionaries**

In Python, the most common way to organize information is in a sequence of values stored in a `list`. A list's natural complement is the dict that stores lookup keys mapped to corresponding values. This chapter covers how to build programs with these versatile building blocks.

## **Chapter 3: Functions**

Functions in Python have a variety of extra features that make a programmer's life easier. Some are similar to capabilities in other programming languages, but many are unique to Python. This chapter covers how to use functions to clarify intention, promote reuse, and reduce bugs.

## **Chapter 4: Comprehensions and Generators**

Python has special syntax for quickly iterating through lists, dictionaries, and sets to generate derivative data structures. It also allows for a stream of iterable values to be incrementally returned by a function. This chapter covers how these features can provide better performance, reduced memory usage, and improved readability.



## Chapter 5: Classes and Interfaces

Python is an object-oriented language. Getting things done in Python often requires writing new classes and defining how they interact through their interfaces and hierarchies. This chapter covers how to use classes to express your intended behaviors with objects.

## Chapter 6: Metaclasses and Attributes

Metaclasses and dynamic attributes are powerful Python features. However, they also enable you to implement extremely bizarre and unexpected behaviors. This chapter covers the common idioms for using these mechanisms to ensure that you follow the *rule of least surprise*.

## Chapter 7: Concurrency and Parallelism

Python makes it easy to write concurrent programs that do many different things seemingly at the same time. Python can also be used to do parallel work through system calls, subprocesses, and C extensions. This chapter covers how to best utilize Python in these subtly different situations.

## Chapter 8: Robustness and Performance

Python has built-in features and modules that aid in hardening your programs so they are dependable. Python also includes tools to help you achieve higher performance with minimal effort. This chapter covers how to use Python to optimize your programs to maximize their reliability and efficiency in production.

## Chapter 9: Testing and Debugging

You should always test your code, regardless of what language it's written in. However, Python's dynamic features can increase the risk of runtime errors in unique ways. Luckily, they also make it easier to write tests and diagnose malfunctioning programs. This chapter covers Python's built-in tools for testing and debugging.

## Chapter 10: Collaboration

Collaborating on Python programs requires you to be deliberate about how you write your code. Even if you're working alone, you'll want to understand how to use modules written by others. This chapter covers the standard tools and best practices that enable people to work together on Python programs.

## Conventions Used in This Book

Python code snippets in this book are in `monospace` font and have syntax highlighting. When lines are long, I use characters to show when they wrap. I truncate some snippets with ellipses (...) to indicate regions where code exists that isn't essential for expressing the point. You'll need to download the full example code (see below on where to get it) to get these truncated snippets to run correctly on your computer.

I take some artistic license with the Python style guide in order to make the code examples better fit the format of a book, or to highlight the most important parts. I've also left out embedded documentation to reduce the size of code examples. I strongly suggest that you don't emulate this in your projects; instead, you should follow the style guide (see [Item 2: “Follow the PEP 8 Style Guide”](#)) and write documentation (see [Item 84: “Write Docstrings for Every Function, Class, and Module”](#)).

Most code snippets in this book are accompanied by the corresponding output from running the code. When I say “output,” I mean console or terminal output: what you see when running the Python program in an interactive interpreter. Output sections are in `monospace` font and are preceded by a `>>>` line (the Python interactive prompt). The idea is that you could type the code snippets into a Python shell and reproduce the expected output.

Finally, there are some other sections in `monospace` font that are not preceded by a `>>>` line. These represent the output of running programs besides the normal Python interpreter. These examples often begin with `$` characters to indicate that I'm running programs from a command-line shell like Bash. If you're running these commands on Windows or another type

of system, you may need to adjust the program names and arguments accordingly.

## **Where to Get the Code and Errata**

It's useful to view some of the examples in this book as whole programs without interleaved prose. This also gives you a chance to tinker with the code yourself and understand why the program works as described. You can find the source code for all code snippets in this book on the book's website at <https://effectivepython.com>. The website also includes any corrections to the book, as well as how to report errors.

# Acknowledgments

This book would not have been possible without the guidance, support, and encouragement from many people in my life.

Thanks to Scott Meyers for the Effective Software Development series. I first read *Effective C++* when I was 15 years old and fell in love with programming. There's no doubt that Scott's books led to my academic experience and first job. I'm thrilled to have had the opportunity to write this book.

Thanks to my technical reviewers for the depth and thoroughness of their feedback for the second edition of this book: Andy Chu, Nick Cohron, Andrew Dolan, Asher Mancinelli, and Alex Martelli. Thanks to my colleagues at Google for their review and input. Without all of your help, this book would have been inscrutable.

Thanks to everyone at Pearson involved in making this second edition a reality. Thanks to my executive editor Debra Williams for being supportive throughout the process. Thanks to the team who were instrumental: development editor Chris Zahn, marketing manager Stephane Nakib, copy editor Catherine Wilson, senior project editor Lori Lyons, and cover designer Chuti Prasertsith.

Thanks to everyone who supported me in creating the first edition of this book: Trina MacDonald, Brett Cannon, Tavis Rudd, Mike Taylor, Leah Culver, Adrian Holovaty, Michael Levine, Marzia Niccolai, Ade Oshineye, Katrina Sostek, Tom Cirtin, Chris Zahn, Olivia Basegio, Stephane Nakib, Stephanie Geels, Julie Nahil, and Toshiaki Kurokawa. Thanks to all of the readers who reported errors and room for improvement. Thanks to all of the translators who made the book available in other languages around the world.

Thanks to the wonderful Python programmers I've known and worked with: Anthony Baxter, Brett Cannon, Wesley Chun, Jeremy Hylton, Alex Martelli, Neal Norwitz, Guido van Rossum, Andy Smith, Greg Stein, and

Ka-Ping Yee. I appreciate your tutelage and leadership. Python has an excellent community, and I feel lucky to be a part of it.

Thanks to my teammates over the years for letting me be the worst player in the band. Thanks to Kevin Gibbs for helping me take risks. Thanks to Ken Ashcraft, Ryan Barrett, and Jon McAlister for showing me how it's done. Thanks to Brad Fitzpatrick for taking it to the next level. Thanks to Paul McDonald for being an amazing co-founder. Thanks to Jeremy Ginsberg, Jack Hebert, John Skidgel, Evan Martin, Tony Chang, Troy Trimble, Tessa Pupius, and Dylan Lorimer for helping me learn. Thanks to Sagnik Nandy and Waleed Ojeil for your mentorship.

Thanks to the inspiring programming and engineering teachers that I've had: Ben Chelf, Glenn Cowan, Vince Hugo, Russ Lewin, Jon Stemmler, Derek Thomson, and Daniel Wang. Without your instruction, I would never have pursued our craft or gained the perspective required to teach others.

Thanks to my mother for giving me a sense of purpose and encouraging me to become a programmer. Thanks to my brother, my grandparents, and the rest of my family and childhood friends for being role models as I grew up and found my passion.

Finally, thanks to my wife, Colleen, for her love, support, and laughter through the journey of life.

## About the Author

**Brett Slatkin** is a principal software engineer at Google. He is the technical co-founder of Google Surveys, the co-creator of the PubSubHubbub protocol, and he launched Google's first cloud computing product (App Engine). Fourteen years ago, he cut his teeth using Python to manage Google's enormous fleet of servers.

Outside of his day job, he likes to play piano and surf (both poorly). He also enjoys writing about programming-related topics on his personal website (<https://onebigfluke.com>). He earned his B.S. in computer engineering from Columbia University in the City of New York. He lives in San Francisco.

# 1. Pythonic Thinking

The idioms of a programming language are defined by its users. Over the years, the Python community has come to use the adjective *Pythonic* to describe code that follows a particular style. The Pythonic style isn't regimented or enforced by the compiler. It has emerged over time through experience using the language and working with others. Python programmers prefer to be explicit, to choose simple over complex, and to maximize readability. (Type `import this` into your interpreter to read *The Zen of Python*.)

Programmers familiar with other languages may try to write Python as if it's C++, Java, or whatever they know best. New programmers may still be getting comfortable with the vast range of concepts that can be expressed in Python. It's important for you to know the best—the *Pythonic*—way to do the most common things in Python. These patterns will affect every program you write.

## Item 1: Know Which Version of Python You're Using

Throughout this book, the majority of example code is in the syntax of Python 3.7 (released in June 2018). This book also provides some examples in the syntax of Python 3.8 (released in October 2019) to highlight new features that will be more widely available soon. This book does not cover Python 2.

Many computers come with multiple versions of the standard CPython runtime preinstalled. However, the default meaning of `python` on the command line may not be clear. `python` is usually an alias for `python2.7`, but it can sometimes be an alias for even older versions, like `python2.6` or `python2.5`. To find out exactly which version of Python you're using, you can use the `--version` flag:

```
$ python --version
Python 2.7.10
```

Python 3 is usually available under the name `python3`:

```
$ python3 --version
Python 3.8.0
```

You can also figure out the version of Python you’re using at runtime by inspecting values in the `sys` built-in module:

[Click here to view code image](#)

```
import sys
print(sys.version_info)
print(sys.version)

>>>
sys.version_info(major=3, minor=8, micro=0,
➔releaselevel='final', serial=0)
3.8.0 (default, Oct 21 2019, 12:51:32)
[Clang 6.0 (clang-600.0.57)]
```

Python 3 is actively maintained by the Python core developers and community, and it is constantly being improved. Python 3 includes a variety of powerful new features that are covered in this book. The majority of Python’s most common open source libraries are compatible with and focused on Python 3. I strongly encourage you to use Python 3 for all your Python projects.

Python 2 is scheduled for *end of life* after January 1, 2020, at which point all forms of bug fixes, security patches, and backports of features will cease. Using Python 2 after that date is a liability because it will no longer be officially maintained. If you’re still stuck working in a Python 2 codebase, you should consider using helpful tools like `2to3` (preinstalled with Python) and `six` (available as a community package; see [Item 82](#): “[Know Where to Find Community-Built Modules](#)”) to help you make the transition to Python 3.

## Things to Remember

- ◆ Python 3 is the most up-to-date and well-supported version of Python, and you should use it for your projects.



- ◆ Be sure that the command-line executable for running Python on your system is the version you expect it to be.
- ◆ Avoid Python 2 because it will no longer be maintained after January 1, 2020.

## Item 2: Follow the PEP 8 Style Guide

Python Enhancement Proposal #8, otherwise known as PEP 8, is the style guide for how to format Python code. You are welcome to write Python code any way you want, as long as it has valid syntax. However, using a consistent style makes your code more approachable and easier to read. Sharing a common style with other Python programmers in the larger community facilitates collaboration on projects. But even if you are the only one who will ever read your code, following the style guide will make it easier for you to change things later, and can help you avoid many common errors.

PEP 8 provides a wealth of details about how to write clear Python code. It continues to be updated as the Python language evolves. It's worth reading the whole guide online (<https://www.python.org/dev/peps/pep-0008/>). Here are a few rules you should be sure to follow.

### Whitespace

In Python, whitespace is syntactically significant. Python programmers are especially sensitive to the effects of whitespace on code clarity. Follow these guidelines related to whitespace:

- Use spaces instead of tabs for indentation.
- Use four spaces for each level of syntactically significant indenting.
- Lines should be 79 characters in length or less.
- Continuations of long expressions onto additional lines should be indented by four extra spaces from their normal indentation level.
- In a file, functions and classes should be separated by two blank lines.

- In a class, methods should be separated by one blank line.
- In a dictionary, put no whitespace between each key and colon, and put a single space before the corresponding value if it fits on the same line.
- Put one—and only one—space before and after the = operator in a variable assignment.
- For type annotations, ensure that there is no separation between the variable name and the colon, and use a space before the type information.

## Naming

PEP 8 suggests unique styles of naming for different parts in the language. These conventions make it easy to distinguish which type corresponds to each name when reading code. Follow these guidelines related to naming:

- Functions, variables, and attributes should be in `lowercase_underscore` format.
- Protected instance attributes should be in `_leading_underscore` format.
- Private instance attributes should be in `__double_leading_underscore` format.
- Classes (including exceptions) should be in `CapitalizedWord` format.
- Module-level constants should be in `ALL_CAPS` format.
- Instance methods in classes should use `self`, which refers to the object, as the name of the first parameter.
- Class methods should use `cls`, which refers to the class, as the name of the first parameter.

## Expressions and Statements

*The Zen of Python* states: “There should be one—and preferably only one—obvious way to do it.” PEP 8 attempts to codify this style in its guidance for expressions and statements:

- Use inline negation (`if a is not b`) instead of negation of positive expressions (`if not a is b`).
- Don't check for empty containers or sequences (like `[]` or `''`) by comparing the length to zero (`if len(somelist) == 0`). Use `if not somelist` and assume that empty values will implicitly evaluate to `False`.
- The same thing goes for non-empty containers or sequences (like `[1]` or `'hi'`). The statement `if somelist` is implicitly `True` for non-empty values.
- Avoid single-line `if` statements, `for` and `while` loops, and `except` compound statements. Spread these over multiple lines for clarity.
- If you can't fit an expression on one line, surround it with parentheses and add line breaks and indentation to make it easier to read.
- Prefer surrounding multiline expressions with parentheses over using the `\` line continuation character.

## Imports

PEP 8 suggests some guidelines for how to import modules and use them in your code:

- Always put `import` statements (including `from x import y`) at the top of a file.
- Always use absolute names for modules when importing them, not names relative to the current module's own path. For example, to import the `foo` module from within the `bar` package, you should use `from bar import foo`, not just `import foo`.
- If you must do relative imports, use the explicit syntax `from . import foo`.
- Imports should be in sections in the following order: standard library modules, third-party modules, your own modules. Each subsection should have imports in alphabetical order.

## Note

The Pylint tool (<https://www.pylint.org>) is a popular static analyzer for Python source code. Pylint provides automated enforcement of the PEP 8 style guide and detects many other types of common errors in Python programs. Many IDEs and editors also include linting tools or support similar plug-ins.

## Things to Remember

- ✦ Always follow the Python Enhancement Proposal #8 (PEP 8) style guide when writing Python code.
- ✦ Sharing a common style with the larger Python community facilitates collaboration with others.
- ✦ Using a consistent style makes it easier to modify your own code later.

## Item 3: Know the Differences Between `bytes` and `str`

In Python, there are two types that represent sequences of character data: `bytes` and `str`. Instances of `bytes` contain raw, unsigned 8-bit values (often displayed in the ASCII encoding):

```
a = b'h\x65llo'
print(list(a))
print(a)
```

```
>>>
[104, 101, 108, 108, 111]
b'hello'
```

Instances of `str` contain Unicode *code points* that represent textual characters from human languages:

[Click here to view code image](#)

```
a = 'a\u0300 propos'
print(list(a))
print(a)
```

```
>>>
```

```
['a', ' ', ' ', 'p', 'r', 'o', 'p', 'o', 's']  
à propos
```

Importantly, `str` instances do not have an associated binary encoding, and `bytes` instances do not have an associated text encoding. To convert Unicode data to binary data, you must call the `encode` method of `str`. To convert binary data to Unicode data, you must call the `decode` method of `bytes`. You can explicitly specify the encoding you want to use for these methods, or accept the system default, which is commonly *UTF-8* (but not always—see more on that below).

When you’re writing Python programs, it’s important to do encoding and decoding of Unicode data at the furthest boundary of your interfaces; this approach is often called the *Unicode sandwich*. The core of your program should use the `str` type containing Unicode data and should not assume anything about character encodings. This approach allows you to be very accepting of alternative text encodings (such as *Latin-1*, *Shift JIS*, and *Big5*) while being strict about your output text encoding (ideally, *UTF-8*).

The split between character types leads to two common situations in Python code:

- You want to operate on raw 8-bit sequences that contain *UTF-8*-encoded strings (or some other encoding).
- You want to operate on Unicode strings that have no specific encoding.

You’ll often need two helper functions to convert between these cases and to ensure that the type of input values matches your code’s expectations.

The first function takes a `bytes` or `str` instance and always returns a `str`:

[Click here to view code image](#)

```
def to_str(bytes_or_str):  
    if isinstance(bytes_or_str, bytes):  
        value = bytes_or_str.decode('utf-8')  
    else:  
        value = bytes_or_str  
    return value # Instance of str
```

```
print(repr(to_str(b'foo')))
print(repr(to_str('bar')))
```

```
>>>
'foo'
'bar'
```

The second function takes a bytes or str instance and always returns a bytes:

[Click here to view code image](#)

```
def to_bytes(bytes_or_str):
    if isinstance(bytes_or_str, str):
        value = bytes_or_str.encode('utf-8')
    else:
        value = bytes_or_str
    return value # Instance of bytes
print(repr(to_bytes(b'foo')))
print(repr(to_bytes('bar')))
```

There are two big gotchas when dealing with raw 8-bit values and Unicode strings in Python.

The first issue is that bytes and str seem to work the same way, but their instances are not compatible with each other, so you must be deliberate about the types of character sequences that you're passing around.

By using the + operator, you can add bytes to bytes and str to str, respectively:

```
print(b'one' + b'two')
print('one' + 'two')
```

```
>>>
b'onetwo'
onetwo
```

But you can't add str instances to bytes instances:

[Click here to view code image](#)

```
b'one' + 'two'
```

```
>>>
```

```
Traceback ...
TypeError: can't concat str to bytes
```

Nor can you add bytes instances to str instances:

[Click here to view code image](#)

```
'one' + b'two'
```

```
>>>
Traceback ...
TypeError: can only concatenate str (not "bytes") to str
```

By using binary operators, you can compare bytes to bytes and str to str, respectively:

```
assert b'red' > b'blue'
assert 'red' > 'blue'
```

But you can't compare a str instance to a bytes instance:

[Click here to view code image](#)

```
assert 'red' > b'blue'
```

```
>>>
Traceback ...
TypeError: '>' not supported between instances of 'str' and
➡'bytes'
```

Nor can you compare a bytes instance to a str instance:

[Click here to view code image](#)

```
assert b'blue' < 'red'
```

```
>>>
Traceback ...
TypeError: '<' not supported between instances of 'bytes'
➡and 'str'
```

Comparing bytes and str instances for equality will always evaluate to False, even when they contain exactly the same characters (in this case, ASCII-encoded “foo”):

```
print(b'foo' == 'foo')
```

```
>>>
False
```

The % operator works with format strings for each type, respectively:

```
print(b'red %s' % b'blue')
print('red %s' % 'blue')
```

```
>>>
b'red blue'
red blue
```

But you can't pass a str instance to a bytes format string because Python doesn't know what binary text encoding to use:

[Click here to view code image](#)

```
print(b'red %s' % 'blue')
```

```
>>>
Traceback ...
TypeError: %b requires a bytes-like object, or an object that
↳ implements __bytes__, not 'str'
```

You *can* pass a bytes instance to a str format string using the % operator, but it doesn't do what you'd expect:

```
print('red %s' % b'blue')
```

```
>>>
red b'blue'
```

This code actually invokes the `__repr__` method (see [Item 75: “Use repr Strings for Debugging Output”](#)) on the bytes instance and substitutes that in place of the %s, which is why `b'blue'` remains escaped in the output.

The second issue is that operations involving file handles (returned by the `open` built-in function) default to requiring Unicode strings instead of raw bytes. This can cause surprising failures, especially for programmers accustomed to Python 2. For example, say that I want to write some binary data to a file. This seemingly simple code breaks:

[Click here to view code image](#)



```
with open('data.bin', 'w') as f:
    f.write(b'\xf1\xf2\xf3\xf4\xf5')
```

```
>>>
```

```
Traceback ...
```

```
TypeError: write() argument must be str, not bytes
```

The cause of the exception is that the file was opened in write text mode ('w') instead of write binary mode ('wb'). When a file is in text mode, write operations expect str instances containing Unicode data instead of bytes instances containing binary data. Here, I fix this by changing the open mode to 'wb':

```
with open('data.bin', 'wb') as f:
    f.write(b'\xf1\xf2\xf3\xf4\xf5')
```

A similar problem also exists for reading data from files. For example, here I try to read the binary file that was written above:

[Click here to view code image](#)

```
with open('data.bin', 'r') as f:
    data = f.read()
```

```
>>>
```

```
Traceback ...
```

```
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xf1 in
➡position 0: invalid continuation byte
```

This fails because the file was opened in read text mode ('r') instead of read binary mode ('rb'). When a handle is in text mode, it uses the system's default text encoding to interpret binary data using the bytes.encode (for writing) and str.decode (for reading) methods. On most systems, the default encoding is UTF-8, which can't accept the binary data b'\xf1\xf2\xf3\xf4\xf5', thus causing the error above. Here, I solve this problem by changing the open mode to 'rb':

[Click here to view code image](#)

```
with open('data.bin', 'rb') as f:
    data = f.read()
```

```
assert data == b'\xf1\xf2\xf3\xf4\xf5'
```

Alternatively, I can explicitly specify the encoding parameter to the open function to make sure that I'm not surprised by any platform-specific behavior. For example, here I assume that the binary data in the file was actually meant to be a string encoded as 'cp1252' (a legacy Windows encoding):

[Click here to view code image](#)

```
with open('data.bin', 'r', encoding='cp1252') as f:
    data = f.read()

assert data == 'ñòóõ'
```

The exception is gone, and the string interpretation of the file's contents is very different from what was returned when reading raw bytes. The lesson here is that you should check the default encoding on your system (using `python3 -c 'import locale; print(locale.getpreferredencoding())'`) to understand how it differs from your expectations. When in doubt, you should explicitly pass the encoding parameter to open.

## Things to Remember

- ✦ bytes contains sequences of 8-bit values, and str contains sequences of Unicode code points.
- ✦ Use helper functions to ensure that the inputs you operate on are the type of character sequence that you expect (8-bit values, UTF-8-encoded strings, Unicode code points, etc).
- ✦ bytes and str instances can't be used together with operators (like >, ==, +, and %).
- ✦ If you want to read or write binary data to/from a file, always open the file using a binary mode (like 'rb' or 'wb').
- ✦ If you want to read or write Unicode data to/from a file, be careful about your system's default text encoding. Explicitly pass the encoding parameter to open if you want to avoid surprises.

## Item 4: Prefer Interpolated F-Strings Over C-style Format Strings and `str.format`

Strings are present throughout Python codebases. They're used for rendering messages in user interfaces and command-line utilities. They're used for writing data to files and sockets. They're used for specifying what's gone wrong in Exception details (see [Item 27: “Use Comprehensions Instead of `map` and `filter`”](#)). They're used in debugging (see [Item 80: “Consider Interactive Debugging with `pdb`”](#) and [Item 75: “Use `repr` Strings for Debugging Output”](#)).

*Formatting* is the process of combining predefined text with data values into a single human-readable message that's stored as a string. Python has four different ways of formatting strings that are built into the language and standard library. All but one of them, which is covered last in this item, have serious shortcomings that you should understand and avoid.

The most common way to format a string in Python is by using the % formatting operator. The predefined text template is provided on the left side of the operator in a *format string*. The values to insert into the template are provided as a single value or tuple of multiple values on the right side of the format operator. For example, here I use the % operator to convert difficult-to-read binary and hexadecimal values to integer strings:

[Click here to view code image](#)

```
a = 0b10111011
b = 0xc5f
print('Binary is %d, hex is %d' % (a, b))
```

```
>>>
Binary is 187, hex is 3167
```

The format string uses format specifiers (like %d) as placeholders that will be replaced by values from the right side of the formatting expression. The syntax for format specifiers comes from C's `printf` function, which has been inherited by Python (as well as by other programming languages). Python supports all the usual options you'd expect from `printf`, such as %s, %x, and %f format specifiers, as well as control over decimal places,

padding, fill, and alignment. Many programmers who are new to Python start with C-style format strings because they're familiar and simple to use.

There are four problems with C-style format strings in Python.

The first problem is that if you change the type or order of data values in the tuple on the right side of a formatting expression, you can get errors due to type conversion incompatibility. For example, this simple formatting expression works:

[Click here to view code image](#)

```
key = 'my_var'
value = 1.234
formatted = '%-10s = %.2f' % (key, value)
print(formatted)

>>>
my_var      = 1.23
```

But if you swap key and value, you get an exception at runtime:

[Click here to view code image](#)

```
reordered_tuple = '%-10s = %.2f' % (value, key)

>>>
Traceback ...
TypeError: must be real number, not str
```

Similarly, leaving the right side parameters in the original order but changing the format string results in the same error:

[Click here to view code image](#)

```
reordered_string = '%.2f = %-10s' % (key, value)

>>>
Traceback ...
TypeError: must be real number, not str
```

To avoid this gotcha, you need to constantly check that the two sides of the % operator are in sync; this process is error prone because it must be done manually for every change.

The second problem with C-style formatting expressions is that they become difficult to read when you need to make small modifications to values before formatting them into a string—and this is an extremely common need. Here, I list the contents of my kitchen pantry without making inline changes:

[Click here to view code image](#)

```
pantry = [
    ('avocados', 1.25),
    ('bananas', 2.5),
    ('cherries', 15),
]
for i, (item, count) in enumerate(pantry):
    print('#%d: %-10s = %.2f' % (i, item, count))

>>>
#0: avocados      = 1.25
#1: bananas       = 2.50
#2: cherries      = 15.00
```

Now, I make a few modifications to the values that I'm formatting to make the printed message more useful. This causes the `tuple` in the formatting expression to become so long that it needs to be split across multiple lines, which hurts readability:

[Click here to view code image](#)

```
for i, (item, count) in enumerate(pantry):
    print('#%d: %-10s = %d' % (
        i + 1,
        item.title(),
        round(count)))

>>>
#1: Avocados      = 1
#2: Bananas       = 2
#3: Cherries      = 15
```

The third problem with formatting expressions is that if you want to use the same value in a format string multiple times, you have to repeat it in the right side tuple:

[Click here to view code image](#)

```
template = '%s loves food. See %s cook.'  
name = 'Max'  
formatted = template % (name, name)  
print(formatted)
```

```
>>>
```

```
Max loves food. See Max cook.
```

This is especially annoying and error prone if you have to repeat small modifications to the values being formatted. For example, here I remembered to call the `title()` method multiple times, but I could have easily added the method call to one reference to `name` and not the other, which would cause mismatched output:

[Click here to view code image](#)

```
name = 'brad'  
formatted = template % (name.title(), name.title())  
print(formatted)
```

```
>>>
```

```
Brad loves food. See Brad cook.
```

To help solve some of these problems, the `%` operator in Python has the ability to also do formatting with a dictionary instead of a `tuple`. The keys from the dictionary are matched with format specifiers with the corresponding name, such as `%(key)s`. Here, I use this functionality to change the order of values on the right side of the formatting expression with no effect on the output, thus solving problem #1 from above:

[Click here to view code image](#)

```
key = 'my_var'  
value = 1.234  
  
old_way = '%-10s = %.2f' % (key, value)  
  
new_way = '%(key)-10s = %(value).2f' % {  
    'key': key, 'value': value} # Original  
  
reordered = '%(key)-10s = %(value).2f' % {  
    'value': value, 'key': key} # Swapped  
  
assert old_way == new_way == reordered
```

Using dictionaries in formatting expressions also solves problem #3 from above by allowing multiple format specifiers to reference the same value, thus making it unnecessary to supply that value more than once:

[Click here to view code image](#)

```
name = 'Max'
```

```
template = '%s loves food. See %s cook.'  
before = template % (name, name) # Tuple
```

```
template = '%(name)s loves food. See %(name)s cook.'  
after = template % {'name': name} # Dictionary
```

```
assert before == after
```

However, dictionary format strings introduce and exacerbate other issues. For problem #2 above, regarding small modifications to values before formatting them, formatting expressions become longer and more visually noisy because of the presence of the dictionary key and colon operator on the right side. Here, I render the same string with and without dictionaries to show this problem:

[Click here to view code image](#)

```
for i, (item, count) in enumerate(pantry):  
    before = '#%d: %-10s = %d' % (  
        i + 1,  
        item.title(),  
        round(count))  
  
    after = '#%(loop)d: %(item)-10s = %(count)d' % {  
        'loop': i + 1,  
        'item': item.title(),  
        'count': round(count),  
    }  
  
    assert before == after
```

Using dictionaries in formatting expressions also increases verbosity, which is problem #4 with C-style formatting expressions in Python. Each key

must be specified at least twice—once in the format specifier, once in the dictionary as a key, and potentially once more for the variable name that contains the dictionary value:

[Click here to view code image](#)

```
soup = 'lentil'
formatted = 'Today\'s soup is %(soup)s.' % {'soup': soup}
print(formatted)

>>>
Today's soup is lentil.
```

Besides the duplicative characters, this redundancy causes formatting expressions that use dictionaries to be long. These expressions often must span multiple lines, with the format strings being concatenated across multiple lines and the dictionary assignments having one line per value to use in formatting:

[Click here to view code image](#)

```
menu = {
    'soup': 'lentil',
    'oyster': 'kumamoto',
    'special': 'schnitzel',
}
template = ('Today\'s soup is %(soup)s, '
            'buy one get two %(oyster)s oysters, '
            'and our special entrée is %(special)s.')
formatted = template % menu
print(formatted)

>>>
Today's soup is lentil, buy one get two kumamoto oysters, and
our special entrée is schnitzel.
```

To understand what this formatting expression is going to produce, your eyes have to keep going back and forth between the lines of the format string and the lines of the dictionary. This disconnect makes it hard to spot bugs, and readability gets even worse if you need to make small modifications to any of the values before formatting.

There must be a better way.



## The `format` Built-in and `str.format`

Python 3 added support for *advanced string formatting* that is more expressive than the old C-style format strings that use the `%` operator. For individual Python values, this new functionality can be accessed through the `format` built-in function. For example, here I use some of the new options (`,` for thousands separators and `^` for centering) to format values:

```
a = 1234.5678
formatted = format(a, ',.2f')
print(formatted)

b = 'my string'
formatted = format(b, '^20s')
print('*', formatted, '*')

>>>
1,234.57
*      my string      *
```

You can use this functionality to format multiple values together by calling the new `format` method of the `str` type. Instead of using C-style format specifiers like `%d`, you can specify placeholders with `{}`. By default the placeholders in the format string are replaced by the corresponding positional arguments passed to the `format` method in the order in which they appear:

[Click here to view code image](#)

```
key = 'my_var'
value = 1.234

formatted = '{} = {}'.format(key, value)
print(formatted)

>>>
my_var = 1.234
```

Within each placeholder you can optionally provide a colon character followed by format specifiers to customize how values will be converted into strings (see `help('FORMATTING')` for the full range of options):

[Click here to view code image](#)

```
formatted = '{:<10} = {:.2f}'.format(key, value)
print(formatted)
```

```
>>>
my_var      = 1.23
```

The way to think about how this works is that the format specifiers will be passed to the `format` built-in function along with the value (`format(value, '.2f')` in the example above). The result of that function call is what replaces the placeholder in the overall formatted string. The formatting behavior can be customized per class using the `__format__` special method.

With C-style format strings, you need to escape the `%` character (by doubling it) so it's not interpreted as a placeholder accidentally. With the `str.format` method you need to similarly escape braces:

[Click here to view code image](#)

```
print('%0.2f%' % 12.5)
print('{} replaces {}'.format(1.23))
```

```
>>>
12.50%
1.23 replaces {}
```

Within the braces you may also specify the positional index of an argument passed to the `format` method to use for replacing the placeholder. This allows the format string to be updated to reorder the output without requiring you to also change the right side of the formatting expression, thus addressing problem #1 from above:

[Click here to view code image](#)

```
formatted = '{1} = {0}'.format(key, value)
print(formatted)
```

```
>>>
1.234 = my_var
```

The same positional index may also be referenced multiple times in the format string without the need to pass the value to the `format` method more than once, which solves problem #3 from above:

[Click here to view code image](#)

```
formatted = '{0} loves food. See {0} cook.'.format(name)
print(formatted)
```

```
>>>
```

```
Max loves food. See Max cook.
```

Unfortunately, the new `format` method does nothing to address problem #2 from above, leaving your code difficult to read when you need to make small modifications to values before formatting them. There's little difference in readability between the old and new options, which are similarly noisy:

[Click here to view code image](#)

```
for i, (item, count) in enumerate(pantry):
    old_style = '#%d: %-10s = %d' % (
        i + 1,
        item.title(),
        round(count))
    new_style = '#{ }: {:<10s} = {}'.format(
        i + 1,
        item.title(),
        round(count))

    assert old_style == new_style
```

There are even more advanced options for the specifiers used with the `str.format` method, such as using combinations of dictionary keys and list indexes in placeholders, and coercing values to Unicode and repr strings:

[Click here to view code image](#)

```
formatted = 'First letter is {menu[oyster][0]!r}'.format(
    menu=menu)
print(formatted)
```

```
>>>
```

```
First letter is 'k'
```

But these features don't help reduce the redundancy of repeated keys from problem #4 above. For example, here I compare the verbosity of using

dictionaries in C-style formatting expressions to the new style of passing keyword arguments to the `format` method:

[Click here to view code image](#)

```
old_template = (
    'Today\'s soup is %(soup)s, '
    'buy one get two %(oyster)s oysters, '
    'and our special entrée is %(special)s.')
old_formatted = template % {
    'soup': 'lentil',
    'oyster': 'kumamoto',
    'special': 'schnitzel',
}

new_template = (
    'Today\'s soup is {soup}, '
    'buy one get two {oyster} oysters, '
    'and our special entrée is {special}.')
new_formatted = new_template.format(
    soup='lentil',
    oyster='kumamoto',
    special='schnitzel',
)

assert old_formatted == new_formatted
```

This style is slightly less noisy because it eliminates some quotes in the dictionary and a few characters in the format specifiers, but it's hardly compelling. Further, the advanced features of using dictionary keys and indexes within placeholders only provides a tiny subset of Python's expression functionality. This lack of expressiveness is so limiting that it undermines the value of the `format` method from `str` overall.

Given these shortcomings and the problems from C-style formatting expressions that remain (problems #2 and #4 from above), I suggest that you avoid the `str.format` method in general. It's important to know about the new mini language used in format specifiers (everything after the colon) and how to use the `format` built-in function. But the rest of the `str.format` method should be treated as a historical artifact to help you understand how Python's new *f-strings* work and why they're so great.

## Interpolated Format Strings

Python 3.6 added *interpolated format strings*—*f-strings* for short—to solve these issues once and for all. This new language syntax requires you to prefix format strings with an `f` character, which is similar to how byte strings are prefixed with a `b` character and raw (unescaped) strings are prefixed with an `r` character.

F-strings take the expressiveness of format strings to the extreme, solving problem #4 from above by completely eliminating the redundancy of providing keys and values to be formatted. They achieve this pithiness by allowing you to reference all names in the current Python scope as part of a formatting expression:

```
key = 'my_var'
value = 1.234

formatted = f'{key} = {value}'
print(formatted)

>>>
my_var = 1.234
```

All of the same options from the new format built-in mini language are available after the colon in the placeholders within an f-string, as is the ability to coerce values to Unicode and `repr` strings similar to the `str.format` method:

[Click here to view code image](#)

```
formatted = f'{key!r:<10} = {value:.2f}'
print(formatted)

>>>
'my_var' = 1.23
```

Formatting with f-strings is shorter than using C-style format strings with the `%` operator and the `str.format` method in all cases. Here, I show all these options together in order of shortest to longest, and line up the left side of the assignment so you can easily compare them:

[Click here to view code image](#)

```

f_string = f'{key:<10} = {value:.2f}'

c_tuple  = '%-10s = %.2f' % (key, value)

str_args = '{:<10} = {:.2f}'.format(key, value)

str_kw   = '{key:<10} = {value:.2f}'.format(key=key,
                                           value=value)

c_dict   = '%(key)-10s = %(value).2f' % {'key': key,
                                         'value': value}

assert c_tuple == c_dict == f_string
assert str_args == str_kw == f_string

```

F-strings also enable you to put a full Python expression within the placeholder braces, solving problem #2 from above by allowing small modifications to the values being formatted with concise syntax. What took multiple lines with C-style formatting and the `str.format` method now easily fits on a single line:

[Click here to view code image](#)

```

for i, (item, count) in enumerate(pantry):
    old_style = '#%d: %-10s = %d' % (
        i + 1,
        item.title(),
        round(count))

    new_style = '#{i+1}: {item.title():<10s} = {round(count)}'.format(
        i + 1,
        item.title(),
        round(count))

    f_string = f'#{i+1}: {item.title():<10s} = {round(count)}'

    assert old_style == new_style == f_string

```

Or, if it's clearer, you can split an f-string over multiple lines by relying on adjacent-string concatenation (similar to C). Even though this is longer than the single-line version, it's still much clearer than any of the other multiline approaches:

[Click here to view code image](#)

```

for i, (item, count) in enumerate(pantry):
    print(f'#{i+1}: '
          f'{item.title():<10s} = '
          f'{round(count)}')

>>>
#1: Avocados    = 1
#2: Bananas     = 2
#3: Cherries    = 15

```

Python expressions may also appear within the format specifier options. For example, here I parameterize the number of digits to print by using a variable instead of hard-coding it in the format string:

[Click here to view code image](#)

```

places = 3
number = 1.23456
print(f'My number is {number:.{places}f}')

>>>
My number is 1.235

```

The combination of expressiveness, terseness, and clarity provided by f-strings makes them the best built-in option for Python programmers. Any time you find yourself needing to format values into strings, choose f-strings over the alternatives.

## Things to Remember

- ✦ C-style format strings that use the % operator suffer from a variety of gotchas and verbosity problems.
- ✦ The `str.format` method introduces some useful concepts in its formatting specifiers mini language, but it otherwise repeats the mistakes of C-style format strings and should be avoided.
- ✦ F-strings are a new syntax for formatting values into strings that solves the biggest problems with C-style format strings.
- ✦ F-strings are succinct yet powerful because they allow for arbitrary Python expressions to be directly embedded within format specifiers.

## Item 5: Write Helper Functions Instead of Complex Expressions

Python's pithy syntax makes it easy to write single-line expressions that implement a lot of logic. For example, say that I want to decode the query string from a URL. Here, each query string parameter represents an integer value:

[Click here to view code image](#)

```
from urllib.parse import parse_qs

my_values = parse_qs('red=5&blue=0&green=',
                    keep_blank_values=True)
print(repr(my_values))

>>>
{'red': ['5'], 'blue': ['0'], 'green': ['']}
```

Some query string parameters may have multiple values, some may have single values, some may be present but have blank values, and some may be missing entirely. Using the `get` method on the result dictionary will return different values in each circumstance:

[Click here to view code image](#)

```
print('Red:      ', my_values.get('red'))
print('Green:    ', my_values.get('green'))
print('Opacity:  ', my_values.get('opacity'))

>>>
Red:      ['5']
Green:    ['']
Opacity:  None
```

It'd be nice if a default value of `0` were assigned when a parameter isn't supplied or is blank. I might choose to do this with Boolean expressions because it feels like this logic doesn't merit a whole `if` statement or helper function quite yet.

Python's syntax makes this choice all too easy. The trick here is that the empty string, the empty `list`, and zero all evaluate to `False` implicitly.



Thus, the expressions below will evaluate to the subexpression after the `or` operator when the first subexpression is `False`:

[Click here to view code image](#)

```
# For query string 'red=5&blue=0&green='
red = my_values.get('red', [''])[0] or 0
green = my_values.get('green', [''])[0] or 0
opacity = my_values.get('opacity', [''])[0] or 0
print(f'Red:      {red!r}')
print(f'Green:    {green!r}')
print(f'Opacity:  {opacity!r}')

>>>
Red:      '5'
Green:    0
Opacity:  0
```

The `red` case works because the key is present in the `my_values` dictionary. The value is a `list` with one member: the string `'5'`. This string implicitly evaluates to `True`, so `red` is assigned to the first part of the `or` expression.

The `green` case works because the value in the `my_values` dictionary is a `list` with one member: an empty string. The empty string implicitly evaluates to `False`, causing the `or` expression to evaluate to `0`.

The `opacity` case works because the value in the `my_values` dictionary is missing altogether. The behavior of the `get` method is to return its second argument if the key doesn't exist in the dictionary (see [Item 16: “Prefer `get` Over `in` and `KeyError` to Handle Missing Dictionary Keys”\). The default value in this case is a `list` with one member: an empty string. When `opacity` isn't found in the dictionary, this code does exactly the same thing as the `green` case.](#)

However, this expression is difficult to read, and it still doesn't do everything I need. I'd also want to ensure that all the parameter values are converted to integers so I can immediately use them in mathematical expressions. To do that, I'd wrap each expression with the `int` built-in function to parse the string as an integer:

[Click here to view code image](#)

```
red = int(my_values.get('red', [''])[0] or 0)
```

This is now extremely hard to read. There's so much visual noise. The code isn't approachable. A new reader of the code would have to spend too much time picking apart the expression to figure out what it actually does. Even though it's nice to keep things short, it's not worth trying to fit this all on one line.

Python has `if/else` conditional—or ternary—expressions to make cases like this clearer while keeping the code short:

[Click here to view code image](#)

```
red_str = my_values.get('red', [''])
red = int(red_str[0]) if red_str[0] else 0
```

This is better. For less complicated situations, `if/else` conditional expressions can make things very clear. But the example above is still not as clear as the alternative of a full `if/else` statement over multiple lines. Seeing all of the logic spread out like this makes the dense version seem even more complex:

[Click here to view code image](#)

```
green_str = my_values.get('green', [''])
if green_str[0]:
    green = int(green_str[0])
else:
    green = 0
```

If you need to reuse this logic repeatedly—even just two or three times, as in this example—then writing a helper function is the way to go:

[Click here to view code image](#)

```
def get_first_int(values, key, default=0):
    found = values.get(key, [''])

    if found[0]:
        return int(found[0])
    return default
```

The calling code is much clearer than the complex expression using `or` and the two-line version using the `if/else` expression:

[Click here to view code image](#)

```
green = get_first_int(my_values, 'green')
```

As soon as expressions get complicated, it's time to consider splitting them into smaller pieces and moving logic into helper functions. What you gain in readability always outweighs what brevity may have afforded you. Avoid letting Python's pithy syntax for complex expressions from getting you into a mess like this. Follow the *DRY principle*: Don't repeat yourself.

## Things to Remember

- ✦ Python's syntax makes it easy to write single-line expressions that are overly complicated and difficult to read.
- ✦ Move complex expressions into helper functions, especially if you need to use the same logic repeatedly.
- ✦ An `if/else` expression provides a more readable alternative to using the Boolean operators `or` and `and` in expressions.

## Item 6: Prefer Multiple Assignment Unpacking Over Indexing

Python has a built-in `tuple` type that can be used to create immutable, ordered sequences of values. In the simplest case, a `tuple` is a pair of two values, such as keys and values from a dictionary:

[Click here to view code image](#)

```
snack_calories = {  
    'chips': 140,  
    'popcorn': 80,  
    'nuts': 190,  
}  
items = tuple(snack_calories.items())  
print(items)  
  
>>>  
(('chips', 140), ('popcorn', 80), ('nuts', 190))
```

The values in tuples can be accessed through numerical indexes:

```
item = ('Peanut butter', 'Jelly')
first = item[0]
second = item[1]
print(first, 'and', second)
```

```
>>>
Peanut butter and Jelly
```

Once a tuple is created, you can't modify it by assigning a new value to an index:

[Click here to view code image](#)

```
pair = ('Chocolate', 'Peanut butter')
pair[0] = 'Honey'
```

```
>>>
Traceback ...
TypeError: 'tuple' object does not support item assignment
```

Python also has syntax for *unpacking*, which allows for assigning multiple values in a single statement. The patterns that you specify in unpacking assignments look a lot like trying to mutate tuples—which isn't allowed—but they actually work quite differently. For example, if you know that a tuple is a pair, instead of using indexes to access its values, you can assign it to a tuple of two variable names:

[Click here to view code image](#)

```
item = ('Peanut butter', 'Jelly')
first, second = item # Unpacking
print(first, 'and', second)
```

```
>>>
Peanut butter and Jelly
```

Unpacking has less visual noise than accessing the tuple's indexes, and it often requires fewer lines. The same pattern matching syntax of unpacking works when assigning to lists, sequences, and multiple levels of arbitrary iterables within iterables. I don't recommend doing the following in your code, but it's important to know that it's possible and how it works:

[Click here to view code image](#)

```

favorite_snacks = {
    'salty': ('pretzels', 100),
    'sweet': ('cookies', 180),
    'veggie': ('carrots', 20),
}
((type1, (name1, cals1)),
 (type2, (name2, cals2)),
 (type3, (name3, cals3))) = favorite_snacks.items()

print(f'Favorite {type1} is {name1} with {cals1} calories')
print(f'Favorite {type2} is {name2} with {cals2} calories')
print(f'Favorite {type3} is {name3} with {cals3} calories')

>>>
Favorite salty is pretzels with 100 calories
Favorite sweet is cookies with 180 calories
Favorite veggie is carrots with 20 calories

```

Newcomers to Python may be surprised to learn that unpacking can even be used to swap values in place without the need to create temporary variables. Here, I use typical syntax with indexes to swap the values between two positions in a list as part of an ascending order sorting algorithm:

[Click here to view code image](#)

```

def bubble_sort(a):
    for _ in range(len(a)):
        for i in range(1, len(a)):
            if a[i] < a[i-1]:
                temp = a[i]
                a[i] = a[i-1]
                a[i-1] = temp

names = ['pretzels', 'carrots', 'arugula', 'bacon']
bubble_sort(names)
print(names)

>>>
['arugula', 'bacon', 'carrots', 'pretzels']

```

However, with unpacking syntax, it's possible to swap indexes in a single line:

[Click here to view code image](#)

```
def bubble_sort(a):
    for _ in range(len(a)):
        for i in range(1, len(a)):
            if a[i] < a[i-1]:
                a[i-1], a[i] = a[i], a[i-1] # Swap

names = ['pretzels', 'carrots', 'arugula', 'bacon']
bubble_sort(names)
print(names)

>>>
['arugula', 'bacon', 'carrots', 'pretzels']
```

The way this swap works is that the right side of the assignment (`a[i]`, `a[i-1]`) is evaluated first, and its values are put into a new temporary, unnamed tuple (such as `('carrots', 'pretzels')` on the first iteration of the loops). Then, the unpacking pattern from the left side of the assignment (`a[i-1]`, `a[i]`) is used to receive that tuple value and assign it to the variable names `a[i-1]` and `a[i]`, respectively. This replaces `'pretzels'` with `'carrots'` at index 0 and `'carrots'` with `'pretzels'` at index 1. Finally, the temporary unnamed tuple silently goes away.

Another valuable application of unpacking is in the target list of `for` loops and similar constructs, such as comprehensions and generator expressions (see [Item 27: “Use Comprehensions Instead of `map` and `filter`”](#) for those). As an example for contrast, here I iterate over a list of snacks without using unpacking:

[Click here to view code image](#)

```
snacks = [('bacon', 350), ('donut', 240), ('muffin', 190)]
for i in range(len(snacks)):
    item = snacks[i]
    name = item[0]
    calories = item[1]
    print(f'#{i+1}: {name} has {calories} calories')

>>>
#1: bacon has 350 calories
#2: donut has 240 calories
#3: muffin has 190 calories
```

This works, but it's noisy. There are a lot of extra characters required in order to index into the various levels of the `snacks` structure. Here, I achieve the same output by using unpacking along with the `enumerate` built-in function (see [Item 7: “Prefer `enumerate` Over `range`”](#)):

[Click here to view code image](#)

```
for rank, (name, calories) in enumerate(snacks, 1):  
    print(f'#{rank}: {name} has {calories} calories')
```

```
>>>  
#1: bacon has 350 calories  
#2: donut has 240 calories  
#3: muffin has 190 calories
```

This is the Pythonic way to write this type of loop; it's short and easy to understand. There's usually no need to access anything using indexes.

Python provides additional unpacking functionality for `list` construction (see [Item 13: “Prefer Catch-All Unpacking Over Slicing”](#)), function arguments (see [Item 22: “Reduce Visual Noise with Variable Positional Arguments”](#)), keyword arguments (see [Item 23: “Provide Optional Behavior with Keyword Arguments”](#)), multiple return values (see [Item 19: “Never Unpack More Than Three Variables When Functions Return Multiple Values”](#)), and more.

Using unpacking wisely will enable you to avoid indexing when possible, resulting in clearer and more Pythonic code.

## Things to Remember

- ◆ Python has special syntax called unpacking for assigning multiple values in a single statement.
- ◆ Unpacking is generalized in Python and can be applied to any iterable, including many levels of iterables within iterables.
- ◆ Reduce visual noise and increase code clarity by using unpacking to avoid explicitly indexing into sequences.

## Item 7: Prefer `enumerate` Over `range`

The `range` built-in function is useful for loops that iterate over a set of integers:

[Click here to view code image](#)

```
from random import randint

random_bits = 0
for i in range(32):
    if randint(0, 1):
        random_bits |= 1 << i

print(bin(random_bits))

>>>
0b11101000100100000111000010000001
```

When you have a data structure to iterate over, like a list of strings, you can loop directly over the sequence:

[Click here to view code image](#)

```
flavor_list = ['vanilla', 'chocolate', 'pecan', 'strawberry']
for flavor in flavor_list:
    print(f'{flavor} is delicious')

>>>
vanilla is delicious
chocolate is delicious
pecan is delicious
strawberry is delicious
```

Often, you'll want to iterate over a list and also know the index of the current item in the list. For example, say that I want to print the ranking of my favorite ice cream flavors. One way to do it is by using `range`:

```
for i in range(len(flavor_list)):
    flavor = flavor_list[i]
    print(f'{i + 1}: {flavor}')

>>>
1: vanilla
2: chocolate
```



```
3: pecan
4: strawberry
```

This looks clumsy compared with the other examples of iterating over `flavor_list` or `range`. I have to get the length of the `list`. I have to index into the array. The multiple steps make it harder to read.

Python provides the `enumerate` built-in function to address this situation. `enumerate` wraps any iterator with a lazy generator (see [Item 30: “Consider Generators Instead of Returning Lists”](#)). `enumerate` yields pairs of the loop index and the next value from the given iterator. Here, I manually advance the returned iterator with the `next` built-in function to demonstrate what it does:

```
it = enumerate(flavor_list)
print(next(it))
print(next(it))

>>>
(0, 'vanilla')
(1, 'chocolate')
```

Each pair yielded by `enumerate` can be succinctly unpacked in a `for` statement (see [Item 6: “Prefer Multiple Assignment Unpacking Over Indexing”](#) for how that works). The resulting code is much clearer:

[Click here to view code image](#)

```
for i, flavor in enumerate(flavor_list):
    print(f'{i + 1}: {flavor}')

>>>
1: vanilla
2: chocolate
3: pecan
4: strawberry
```

I can make this even shorter by specifying the number from which `enumerate` should begin counting (1 in this case) as the second parameter:

[Click here to view code image](#)

```
for i, flavor in enumerate(flavor_list, 1):
    print(f'{i}: {flavor}')
```

## Things to Remember

- ✦ `enumerate` provides concise syntax for looping over an iterator and getting the index of each item from the iterator as you go.
- ✦ Prefer `enumerate` instead of looping over a range and indexing into a sequence.
- ✦ You can supply a second parameter to `enumerate` to specify the number from which to begin counting (zero is the default).

## Item 8: Use `zip` to Process Iterators in Parallel

Often in Python you find yourself with many lists of related objects. List comprehensions make it easy to take a source list and get a derived list by applying an expression (see [Item 27: “Use Comprehensions Instead of `map` and `filter`”](#)):

[Click here to view code image](#)

```
names = ['Cecilia', 'Lise', 'Marie']
counts = [len(n) for n in names]
print(counts)
```

```
>>>
[7, 4, 5]
```

The items in the derived list are related to the items in the source list by their indexes. To iterate over both lists in parallel, I can iterate over the length of the names source list:

```
longest_name = None
max_count = 0

for i in range(len(names)):
    count = counts[i]
    if count > max_count:
        longest_name = names[i]
        max_count = count

print(longest_name)
```

```
>>>
Cecilia
```

The problem is that this whole loop statement is visually noisy. The indexes into `names` and `counts` make the code hard to read. Indexing into the arrays by the loop index `i` happens twice. Using `enumerate` (see [Item 7: “Prefer enumerate Over range”](#)) improves this slightly, but it’s still not ideal:

```
for i, name in enumerate(names):
    count = counts[i]
    if count > max_count:
        longest_name = name
        max_count = count
```

To make this code clearer, Python provides the `zip` built-in function. `zip` wraps two or more iterators with a lazy generator. The `zip` generator yields tuples containing the next value from each iterator. These tuples can be unpacked directly within a `for` statement (see [Item 6: “Prefer Multiple Assignment Unpacking Over Indexing”](#)). The resulting code is much cleaner than the code for indexing into multiple lists:

[Click here to view code image](#)

```
for name, count in zip(names, counts):
    if count > max_count:
        longest_name = name
        max_count = count
```

`zip` consumes the iterators it wraps one item at a time, which means it can be used with infinitely long inputs without risk of a program using too much memory and crashing.

However, beware of `zip`’s behavior when the input iterators are of different lengths. For example, say that I add another item to `names` above but forget to update `counts`. Running `zip` on the two input lists will have an unexpected result:

[Click here to view code image](#)

```
names.append('Rosalind')
for name, count in zip(names, counts):
    print(name)
```

```
>>>
Cecilia
Lise
Marie
```

The new item for 'Rosalind' isn't there. Why not? This is just how `zip` works. It keeps yielding tuples until any one of the wrapped iterators is exhausted. Its output is as long as its shortest input. This approach works fine when you know that the iterators are of the same length, which is often the case for derived lists created by list comprehensions.

But in many other cases, the truncating behavior of `zip` is surprising and bad. If you don't expect the lengths of the lists passed to `zip` to be equal, consider using the `zip_longest` function from the `itertools` built-in module instead:

[Click here to view code image](#)

```
import itertools
for name, count in itertools.zip_longest(names, counts):
    print(f'{name}: {count}')
```

```
>>>
Cecilia: 7 Lise: 4
Marie: 5
Rosalind: None
```

`zip_longest` replaces missing values—the length of the string 'Rosalind' in this case—with whatever `fillvalue` is passed to it, which defaults to `None`.

## Things to Remember

- ♦ The `zip` built-in function can be used to iterate over multiple iterators in parallel.
- ♦ `zip` creates a lazy generator that produces tuples, so it can be used on infinitely long inputs.
- ♦ `zip` truncates its output silently to the shortest iterator if you supply it with iterators of different lengths.

- ✦ Use the `zip_longest` function from the `itertools` built-in module if you want to use `zip` on iterators of unequal lengths without truncation.

## Item 9: Avoid `else` Blocks After `for` and `while` Loops

Python loops have an extra feature that is not available in most other programming languages: You can put an `else` block immediately after a loop's repeated interior block:

```
for i in range(3):
    print('Loop', i)
else:
    print('Else block!')
```

```
>>>
Loop 0 Loop 1
Loop 2
Else block!
```

Surprisingly, the `else` block runs immediately after the loop finishes. Why is the clause called “else”? Why not “and”? In an `if/else` statement, `else` means “Do this if the block before this doesn’t happen.” In a `try/except` statement, `except` has the same definition: “Do this if trying the block before this failed.”

Similarly, `else` from `try/except/else` follows this pattern (see [Item 65: “Take Advantage of Each Block in `try/except/else/finally`”](#)) because it means “Do this if there was no exception to handle.” `try/finally` is also intuitive because it means “Always do this after trying the block before.”

Given all the uses of `else`, `except`, and `finally` in Python, a new programmer might assume that the `else` part of `for/else` means “Do this if the loop wasn’t completed.” In reality, it does exactly the opposite. Using a `break` statement in a loop actually skips the `else` block:

```
for i in range(3):
    print('Loop', i)
    if i == 1:
        break
else:
    print('Else block!')
```

```
>>>
Loop 0
Loop 1
```

Another surprise is that the `else` block runs immediately if you loop over an empty sequence:

```
for x in []:
    print('Never runs')
else:
    print('For Else block!')
```

```
>>>
For Else block!
```

The `else` block also runs when `while` loops are initially `False`:

```
while False:
    print('Never runs')
else:
    print('While Else block!')
```

```
>>>
While Else block!
```

The rationale for these behaviors is that `else` blocks after loops are useful when using loops to search for something. For example, say that I want to determine whether two numbers are coprime (that is, their only common divisor is 1). Here, I iterate through every possible common divisor and test the numbers. After every option has been tried, the loop ends. The `else` block runs when the numbers are coprime because the loop doesn't encounter a `break`:

```
a = 4
b = 9

for i in range(2, min(a, b) + 1):
    print('Testing', i)
    if a % i == 0 and b % i == 0:
        print('Not coprime')
        break
else:
    print('Coprime')
```

```
>>>
Testing 2
Testing 3
Testing 4
Coprime
```

In practice, I wouldn't write the code this way. Instead, I'd write a helper function to do the calculation. Such a helper function is written in two common styles.

The first approach is to return early when I find the condition I'm looking for. I return the default outcome if I fall through the loop:

[Click here to view code image](#)

```
def coprime(a, b):
    for i in range(2, min(a, b) + 1):
        if a % i == 0 and b % i == 0:
            return False
    return True

assert coprime(4, 9)
assert not coprime(3, 6)
```

The second way is to have a result variable that indicates whether I've found what I'm looking for in the loop. I break out of the loop as soon as I find something:

[Click here to view code image](#)

```
def coprime_alternate(a, b):
    is_coprime = True
    for i in range(2, min(a, b) + 1):
        if a % i == 0 and b % i == 0:
            is_coprime = False
            break
    return is_coprime

assert coprime_alternate(4, 9)
assert not coprime_alternate(3, 6)
```

Both approaches are much clearer to readers of unfamiliar code. Depending on the situation, either may be a good choice. However, the expressivity you gain from the `else` block doesn't outweigh the burden you put on people (including yourself) who want to understand your code in the future.

Simple constructs like loops should be self-evident in Python. You should avoid using `else` blocks after loops entirely.

## Things to Remember

- ✦ Python has special syntax that allows `else` blocks to immediately follow `for` and `while` loop interior blocks.
- ✦ The `else` block after a loop runs only if the loop body did not encounter a `break` statement.
- ✦ Avoid using `else` blocks after loops because their behavior isn't intuitive and can be confusing.

## Item 10: Prevent Repetition with Assignment Expressions

An assignment expression—also known as the *walrus operator*—is a new syntax introduced in Python 3.8 to solve a long-standing problem with the language that can cause code duplication. Whereas normal assignment statements are written `a = b` and pronounced “a equals b,” these assignments are written `a := b` and pronounced “a *walrus* b” (because `:=` looks like a pair of eyeballs and tusks).

Assignment expressions are useful because they enable you to assign variables in places where assignment statements are disallowed, such as in the conditional expression of an `if` statement. An assignment expression's value evaluates to whatever was assigned to the identifier on the left side of the walrus operator.

For example, say that I have a basket of fresh fruit that I'm trying to manage for a juice bar. Here, I define the contents of the basket:

```
fresh_fruit = {  
    'apple': 10,  
    'banana': 8,  
    'lemon': 5,  
}
```



When a customer comes to the counter to order some lemonade, I need to make sure there is at least one lemon in the basket to squeeze. Here, I do this by retrieving the count of lemons and then using an `if` statement to check for a non-zero value:

```
def make_lemonade(count):  
    ...  
def out_of_stock():  
    ...  
  
count = fresh_fruit.get('lemon', 0)  
if count:  
    make_lemonade(count)  
else:  
    out_of_stock()
```

The problem with this seemingly simple code is that it's noisier than it needs to be. The `count` variable is used only within the first block of the `if` statement. Defining `count` above the `if` statement causes it to appear to be more important than it really is, as if all code that follows, including the `else` block, will need to access the `count` variable, when in fact that is not the case.

This pattern of fetching a value, checking to see if it's non-zero, and then using it is extremely common in Python. Many programmers try to work around the multiple references to `count` with a variety of tricks that hurt readability (see [Item 5: “Write Helper Functions Instead of Complex Expressions”](#) for an example). Luckily, assignment expressions were added to the language to streamline exactly this type of code. Here, I rewrite this example using the walrus operator:

[Click here to view code image](#)

```
if count := fresh_fruit.get('lemon', 0):  
    make_lemonade(count)  
else:  
    out_of_stock()
```

Though this is only one line shorter, it's a lot more readable because it's now clear that `count` is only relevant to the first block of the `if` statement. The assignment expression is first assigning a value to the `count` variable,

and then evaluating that value in the context of the `if` statement to determine how to proceed with flow control. This twostep behavior—assign and then evaluate—is the fundamental nature of the walrus operator.

Lemons are quite potent, so only one is needed for my lemonade recipe, which means a non-zero check is good enough. If a customer orders a cider, though, I need to make sure that I have at least four apples. Here, I do this by fetching the count from the `fruit_basket` dictionary, and then using a comparison in the `if` statement conditional expression:

```
def make_cider(count):  
    ...  
    count = fresh_fruit.get('apple', 0)  
    if count >= 4:  
        make_cider(count)  
    else:  
        out_of_stock()
```

This has the same problem as the lemonade example, where the assignment of `count` puts distracting emphasis on that variable. Here, I improve the clarity of this code by also using the walrus operator:

[Click here to view code image](#)

```
if (count := fresh_fruit.get('apple', 0)) >= 4:  
    make_cider(count)  
else:  
    out_of_stock()
```

This works as expected and makes the code one line shorter. It's important to note how I needed to surround the assignment expression with parentheses to compare it with 4 in the `if` statement. In the lemonade example, no surrounding parentheses were required because the assignment expression stood on its own as a non-zero check; it wasn't a subexpression of a larger expression. As with other expressions, you should avoid surrounding assignment expressions with parentheses when possible.

Another common variation of this repetitive pattern occurs when I need to assign a variable in the enclosing scope depending on some condition, and then reference that variable shortly afterward in a function call. For example, say that a customer orders some banana smoothies. In order to

make them, I need to have at least two bananas' worth of slices, or else an `OutOfBananas` exception will be raised. Here, I implement this logic in a typical way:

[Click here to view code image](#)

```
def slice_bananas(count):
    ...

class OutOfBananas(Exception):
    pass

def make_smoothies(count):
    ...

pieces = 0
count = fresh_fruit.get('banana', 0)
if count >= 2:
    pieces = slice_bananas(count)

try:
    smoothies = make_smoothies(pieces)
except OutOfBananas:
    out_of_stock()
```

The other common way to do this is to put the `pieces = 0` assignment in the `else` block:

[Click here to view code image](#)

```
count = fresh_fruit.get('banana', 0)
if count >= 2:
    pieces = slice_bananas(count)
else:
    pieces = 0

try:
    smoothies = make_smoothies(pieces)
except OutOfBananas:
    out_of_stock()
```

This second approach can feel odd because it means that the `pieces` variable has two different locations—in each block of the `if` statement—where it can be initially defined. This split definition technically works because of Python's scoping rules (see [Item 21: “Know How Closures Interact with](#)

[Variable Scope](#)”), but it isn’t easy to read or discover, which is why many people prefer the construct above, where the `pieces = 0` assignment is first.

The walrus operator can again be used to shorten this example by one line of code. This small change removes any emphasis on the `count` variable. Now, it’s clearer that `pieces` will be important beyond the `if` statement:

[Click here to view code image](#)

```
pieces = 0
if (count := fresh_fruit.get('banana', 0)) >= 2:
    pieces = slice_bananas(count)

try:
    smoothies = make_smoothies(pieces)
except OutOfBananas:
    out_of_stock()
```

Using the walrus operator also improves the readability of splitting the definition of `pieces` across both parts of the `if` statement. It’s easier to trace the `pieces` variable when the `count` definition no longer precedes the `if` statement:

[Click here to view code image](#)

```
if (count := fresh_fruit.get('banana', 0)) >= 2:
    pieces = slice_bananas(count)
else:
    pieces = 0

try:
    smoothies = make_smoothies(pieces)
except OutOfBananas:
    out_of_stock()
```

One frustration that programmers who are new to Python often have is the lack of a flexible switch/case statement. The general style for approximating this type of functionality is to have a deep nesting of multiple `if`, `elif`, and `else` statements.

For example, imagine that I want to implement a system of precedence so that each customer automatically gets the best juice available and doesn’t

have to order. Here, I define logic to make it so banana smoothies are served first, followed by apple cider, and then finally lemonade:

[Click here to view code image](#)

```
count = fresh_fruit.get('banana', 0)
if count >= 2:
    pieces = slice_bananas(count)
    to_enjoy = make_smoothies(pieces)
else:
    count = fresh_fruit.get('apple', 0)
    if count >= 4:
        to_enjoy = make_cider(count)
    else:
        count = fresh_fruit.get('lemon', 0)
        if count:
            to_enjoy = make_lemonade(count)
        else:
            to_enjoy = 'Nothing'
```

Ugly constructs like this are surprisingly common in Python code. Luckily, the walrus operator provides an elegant solution that can feel nearly as versatile as dedicated syntax for switch/case statements:

[Click here to view code image](#)

```
if (count := fresh_fruit.get('banana', 0)) >= 2:
    pieces = slice_bananas(count)
    to_enjoy = make_smoothies(pieces)
elif (count := fresh_fruit.get('apple', 0)) >= 4:
    to_enjoy = make_cider(count)
elif count := fresh_fruit.get('lemon', 0):
    to_enjoy = make_lemonade(count)
else:
    to_enjoy = 'Nothing'
```

The version that uses assignment expressions is only five lines shorter than the original, but the improvement in readability is vast due to the reduction in nesting and indentation. If you ever see such ugly constructs emerge in your code, I suggest that you move them over to using the walrus operator if possible.

Another common frustration of new Python programmers is the lack of a do/while loop construct. For example, say that I want to bottle juice as new

fruit is delivered until there's no fruit remaining. Here, I implement this logic with a `while` loop:

[Click here to view code image](#)

```
def pick_fruit():
    ...

def make_juice(fruit, count):
    ...

bottles = []
fresh_fruit = pick_fruit()
while fresh_fruit:
    for fruit, count in fresh_fruit.items():
        batch = make_juice(fruit, count)
        bottles.extend(batch)
    fresh_fruit = pick_fruit()
```

This is repetitive because it requires two separate `fresh_fruit = pick_fruit()` calls: one before the loop to set initial conditions, and another at the end of the loop to replenish the list of delivered fruit.

A strategy for improving code reuse in this situation is to use the *loop-and-a-half* idiom. This eliminates the redundant lines, but it also undermines the `while` loop's contribution by making it a dumb infinite loop. Now, all of the flow control of the loop depends on the conditional `break` statement:

[Click here to view code image](#)

```
bottles = []
while True:                                # Loop
    fresh_fruit = pick_fruit()
    if not fresh_fruit:                    # And a half
        break

    for fruit, count in fresh_fruit.items():
        batch = make_juice(fruit, count)
        bottles.extend(batch)
```

The walrus operator obviates the need for the loop-and-a-half idiom by allowing the `fresh_fruit` variable to be reassigned and then conditionally evaluated each time through the `while` loop. This solution is short and easy to read, and it should be the preferred approach in your code:

[Click here to view code image](#)

```
bottles = []
while fresh_fruit := pick_fruit():
    for fruit, count in fresh_fruit.items():
        batch = make_juice(fruit, count)
        bottles.extend(batch)
```

There are many other situations where assignment expressions can be used to eliminate redundancy (see [Item 29: “Avoid Repeated Work in Comprehensions by Using Assignment Expressions”](#) for another). In general, when you find yourself repeating the same expression or assignment multiple times within a grouping of lines, it’s time to consider using assignment expressions in order to improve readability.

## Things to Remember

- ✦ Assignment expressions use the walrus operator (`:=`) to both assign and evaluate variable names in a single expression, thus reducing repetition.
- ✦ When an assignment expression is a subexpression of a larger expression, it must be surrounded with parentheses.
- ✦ Although switch/case statements and do/while loops are not available in Python, their functionality can be emulated much more clearly by using assignment expressions.

## 2. Lists and Dictionaries

Many programs are written to automate repetitive tasks that are better suited to machines than to humans. In Python, the most common way to organize this kind of work is by using a sequence of values stored in a `list` type. Lists are extremely versatile and can be used to solve a variety of problems.

A natural complement to lists is the `dict` type, which stores lookup keys mapped to corresponding values (in what is often called an *associative array* or a *hash table*). Dictionaries provide constant time (amortized) performance for assignments and accesses, which means they are ideal for bookkeeping dynamic information.

Python has special syntax and built-in modules that enhance readability and extend the capabilities of lists and dictionaries beyond what you might expect from simple array, vector, and hash table types in other languages.

### Item 11: Know How to Slice Sequences

Python includes syntax for *slicing* sequences into pieces. Slicing allows you to access a subset of a sequence's items with minimal effort. The simplest uses for slicing are the built-in types `list`, `str`, and `bytes`. Slicing can be extended to any Python class that implements the `__getitem__` and `__setitem__` special methods (see [Item 43: “Inherit from `collections.abc` for Custom Container Types](#)”).

The basic form of the slicing syntax is `somelist[start:end]`, where `start` is inclusive and `end` is exclusive:

[Click here to view code image](#)

```
a = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
print('Middle two: ', a[3:5])
print('All but ends:', a[1:7])
```

```
>>>
Middle two:  ['d', 'e']
All but ends: ['b', 'c', 'd', 'e', 'f', 'g']
```



When slicing from the start of a list, you should leave out the zero index to reduce visual noise:

```
assert a[:5] == a[0:5]
```

When slicing to the end of a list, you should leave out the final index because it's redundant:

```
assert a[5:] == a[5:len(a)]
```

Using negative numbers for slicing is helpful for doing offsets relative to the end of a list. All of these forms of slicing would be clear to a new reader of your code:

[Click here to view code image](#)

```
a[:]      # ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
a[:5]     # ['a', 'b', 'c', 'd', 'e']
a[:-1]    # ['a', 'b', 'c', 'd', 'e', 'f', 'g']
a[4:]     #           ['e', 'f', 'g', 'h']
a[-3:]    #           ['f', 'g', 'h']
a[2:5]    #           ['c', 'd', 'e']
a[2:-1]   #           ['c', 'd', 'e', 'f', 'g']
a[-3:-1]  #           ['f', 'g']
```

There are no surprises here, and I encourage you to use these variations.

Slicing deals properly with start and end indexes that are beyond the boundaries of a list by silently omitting missing items. This behavior makes it easy for your code to establish a maximum length to consider for an input sequence:

```
first_twenty_items = a[:20]
last_twenty_items = a[-20:]
```

In contrast, accessing the same index directly causes an exception:

```
a[20]

>>>
Traceback ...
IndexError: list index out of range
```

Note

Beware that indexing a list by a negated variable is one of the few situations in which you can get surprising results from slicing. For example, the expression `somelist[-n:]` will work fine when `n` is greater than one (e.g., `somelist[-3:]`). However, when `n` is zero, the expression `somelist[-0:]` is equivalent to `somelist[:]` and will result in a copy of the original list.

The result of slicing a list is a whole new list. References to the objects from the original list are maintained. Modifying the result of slicing won't affect the original list:

[Click here to view code image](#)

```
b = a[3:]
print('Before: ', b)
b[1] = 99
print('After: ', b)
print('No change:', a)
```

```
>>>
Before:      ['d', 'e', 'f', 'g', 'h']
After:       ['d', 99, 'f', 'g', 'h']
No change:   ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
```

When used in assignments, slices replace the specified range in the original list. Unlike unpacking assignments (such as `a, b = c[:2]`; see [Item 6: “Prefer Multiple Assignment Unpacking Over Indexing”](#)), the lengths of slice assignments don't need to be the same. The values before and after the assigned slice will be preserved. Here, the list shrinks because the replacement list is shorter than the specified slice:

[Click here to view code image](#)

```
print('Before ', a)
a[2:7] = [99, 22, 14]
print('After ', a)
```

```
>>>
Before  ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
After   ['a', 'b', 99, 22, 14, 'h']
```

And here the list grows because the assigned list is longer than the specific slice:

[Click here to view code image](#)

```
print('Before ', a)
a[2:3] = [47, 11]
print('After ', a)

>>>
Before ['a', 'b', 99, 22, 14, 'h']
After  ['a', 'b', 47, 11, 22, 14, 'h']
```

If you leave out both the start and the end indexes when slicing, you end up with a copy of the original list:

```
b = a[:]
assert b == a and b is not a
```

If you assign to a slice with no start or end indexes, you replace the entire contents of the list with a copy of what's referenced (instead of allocating a new list):

[Click here to view code image](#)

```
b = a
print('Before a', a)
print('Before b', b)
a[:] = [101, 102, 103]
assert a is b          # Still the same list object
print('After a ', a)    # Now has different contents
print('After b ', b)    # Same list, so same contents as a

>>>
Before a ['a', 'b', 47, 11, 22, 14, 'h']
Before b ['a', 'b', 47, 11, 22, 14, 'h']
After a  [101, 102, 103]
After b  [101, 102, 103]
```

## Things to Remember

- ◆ Avoid being verbose when slicing: Don't supply 0 for the start index or the length of the sequence for the end index.

- ✦ Slicing is forgiving of start or end indexes that are out of bounds, which means it's easy to express slices on the front or back boundaries of a sequence (like `a[:20]` or `a[-20:]`).
- ✦ Assigning to a list slice replaces that range in the original sequence with what's referenced even if the lengths are different.

## Item 12: Avoid Striding and Slicing in a Single Expression

In addition to basic slicing (see [Item 11: “Know How to Slice Sequences”](#)), Python has special syntax for the stride of a slice in the form `somelist[start:end:stride]`. This lets you take every *n*th item when slicing a sequence. For example, the stride makes it easy to group by even and odd indexes in a list:

[Click here to view code image](#)

```
x = ['red', 'orange', 'yellow', 'green', 'blue', 'purple']
odds = x[::2]
evens = x[1::2]
print(odds)
print(evens)

>>>
['red', 'yellow', 'blue']
['orange', 'green', 'purple']
```

The problem is that the stride syntax often causes unexpected behavior that can introduce bugs. For example, a common Python trick for reversing a byte string is to slice the string with a stride of `-1`:

```
x = b'mongoose'
y = x[::-1]
print(y)

>>>
b'esoonom'
```

This also works correctly for Unicode strings (see [Item 3: “Know the Differences Between bytes and str”](#)):

```
x = '寿司'
y = x[::-1]
print(y)
```

```
>>>
司寿
```

But it will break when Unicode data is encoded as a UTF-8 byte string:

[Click here to view code image](#)

```
w = '寿司'
x = w.encode('utf-8')
y = x[::-1]
z = y.decode('utf-8')
```

```
>>>
Traceback ...
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xb8 in
position 0: invalid start byte
```

Are negative strides besides -1 useful? Consider the following examples:

[Click here to view code image](#)

```
x = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
x[::2]    # ['a', 'c', 'e', 'g']
x[::-2]   # ['h', 'f', 'd', 'b']
```

Here, `::2` means “Select every second item starting at the beginning.”  
Trickier, `::-2` means “Select every second item starting at the end and moving backward.”

What do you think `2::2` means? What about `-2::-2` vs. `-2:2:-2` vs. `2:2:-2`?

[Click here to view code image](#)

```
x[2::2]      # ['c', 'e', 'g']
x[-2::-2]    # ['g', 'e', 'c', 'a']
x[-2:2:-2]   # ['g', 'e']
x[2:2:-2]    # []
```

The point is that the stride part of the slicing syntax can be extremely confusing. Having three numbers within the brackets is hard enough to read because of its density. Then, it’s not obvious when the start and end indexes

come into effect relative to the stride value, especially when the stride is negative.

To prevent problems, I suggest you avoid using a stride along with start and end indexes. If you must use a stride, prefer making it a positive value and omit start and end indexes. If you must use a stride with start or end indexes, consider using one assignment for striding and another for slicing:

[Click here to view code image](#)

```
y = x[::2]    # ['a', 'c', 'e', 'g']  
z = y[1:-1]  # ['c', 'e']
```

Striding and then slicing creates an extra shallow copy of the data. The first operation should try to reduce the size of the resulting slice by as much as possible. If your program can't afford the time or memory required for two steps, consider using the `itertools` built-in module's `islice` method (see [Item 36: “Consider `itertools` for Working with Iterators and Generators”](#)), which is clearer to read and doesn't permit negative values for start, end, or stride.

## Things to Remember

- ◆ Specifying start, end, and stride in a slice can be extremely confusing.
- ◆ Prefer using positive stride values in slices without start or end indexes. Avoid negative stride values if possible.
- ◆ Avoid using start, end, and stride together in a single slice. If you need all three parameters, consider doing two assignments (one to stride and another to slice) or using `islice` from the `itertools` built-in module.

## Item 13: Prefer Catch-All Unpacking Over Slicing

One limitation of basic unpacking (see [Item 6: “Prefer Multiple Assignment Unpacking Over Indexing”](#)) is that you must know the length of the sequences you're unpacking in advance. For example, here I have a `list` of the ages of cars that are being traded in at a dealership. When I try to take

the first two items of the list with basic unpacking, an exception is raised at runtime:

[Click here to view code image](#)

```
car_ages = [0, 9, 4, 8, 7, 20, 19, 1, 6, 15]
car_ages_descending = sorted(car_ages, reverse=True)
oldest, second_oldest = car_ages_descending
```

```
>>>
Traceback ...
ValueError: too many values to unpack (expected 2)
```

Newcomers to Python often rely on indexing and slicing (see [Item 11: “Know How to Slice Sequences”](#)) for this situation. For example, here I extract the oldest, second oldest, and other car ages from a list of at least two items:

[Click here to view code image](#)

```
oldest = car_ages_descending[0]
second_oldest = car_ages_descending[1]
others = car_ages_descending[2:]
print(oldest, second_oldest, others)
```

```
>>>
20 19 [15, 9, 8, 7, 6, 4, 1, 0]
```

This works, but all of the indexing and slicing is visually noisy. In practice, it’s also error prone to divide the members of a sequence into various subsets this way because you’re much more likely to make off-by-one errors; for example, you might change boundaries on one line and forget to update the others.

To better handle this situation, Python also supports catch-all unpacking through a *starred expression*. This syntax allows one part of the unpacking assignment to receive all values that didn’t match any other part of the unpacking pattern. Here, I use a starred expression to achieve the same result as above without indexing or slicing:

[Click here to view code image](#)

```
oldest, second_oldest, *others = car_ages_descending
print(oldest, second_oldest, others)
```

```
>>>
20 19 [15, 9, 8, 7, 6, 4, 1, 0]
```

This code is shorter, easier to read, and no longer has the error-prone brittleness of boundary indexes that must be kept in sync between lines.

A starred expression may appear in any position, so you can get the benefits of catch-all unpacking anytime you need to extract one slice:

[Click here to view code image](#)

```
oldest, *others, youngest = car_ages_descending
print(oldest, youngest, others)

*others, second_youngest, youngest = car_ages_descending
print(youngest, second_youngest, others)
```

```
>>>
20 0 [19, 15, 9, 8, 7, 6, 4, 1]
0 1 [20, 19, 15, 9, 8, 7, 6, 4]
```

However, to unpack assignments that contain a starred expression, you must have at least one required part, or else you'll get a `SyntaxError`. You can't use a catch-all expression on its own:

[Click here to view code image](#)

```
*others = car_ages_descending

>>>
Traceback ...
SyntaxError: starred assignment target must be in a list or
➔tuple
```

You also can't use multiple catch-all expressions in a single-level unpacking pattern:

[Click here to view code image](#)

```
first, *middle, *second_middle, last = [1, 2, 3, 4]

>>>
```



```
Traceback ...
SyntaxError: two starred expressions in assignment
```

But it is possible to use multiple starred expressions in an unpacking assignment statement, as long as they're catch-alls for different parts of the multilevel structure being unpacked. I don't recommend doing the following (see [Item 19: “Never Unpack More Than Three Variables When Functions Return Multiple Values”](#) for related guidance), but understanding it should help you develop an intuition for how starred expressions can be used in unpacking assignments:

[Click here to view code image](#)

```
car_inventory = {
    'Downtown': ('Silver Shadow', 'Pinto', 'DMC'),
    'Airport': ('Skyline', 'Viper', 'Gremlin', 'Nova'),
}

((loc1, (best1, *rest1)),
 (loc2, (best2, *rest2))) = car_inventory.items()
print(f'Best at {loc1} is {best1}, {len(rest1)} others')
print(f'Best at {loc2} is {best2}, {len(rest2)} others')

>>>
Best at Downtown is Silver Shadow, 2 others
Best at Airport is Skyline, 3 others
```

Starred expressions become `list` instances in all cases. If there are no leftover items from the sequence being unpacked, the catch-all part will be an empty `list`. This is especially useful when you're processing a sequence that you know in advance has at least  $N$  elements:

```
short_list = [1, 2]
first, second, *rest = short_list
print(first, second, rest)

>>>
1 2 []
```

You can also unpack arbitrary iterators with the unpacking syntax. This isn't worth much with a basic multiple-assignment statement. For example, here I unpack the values from iterating over a range of length 2. This

doesn't seem useful because it would be easier to just assign to a static list that matches the unpacking pattern (e.g., [1, 2]):

```
it = iter(range(1, 3))
first, second = it
print(f'{first} and {second}')

>>>
1 and 2
```

But with the addition of starred expressions, the value of unpacking iterators becomes clear. For example, here I have a generator that yields the rows of a CSV file containing all car orders from the dealership this week:

[Click here to view code image](#)

```
def generate_csv():
    yield ('Date', 'Make', 'Model', 'Year', 'Price')
    ...
```

Processing the results of this generator using indexes and slices is fine, but it requires multiple lines and is visually noisy:

[Click here to view code image](#)

```
all_csv_rows = list(generate_csv())
header = all_csv_rows[0]
rows = all_csv_rows[1:]
print('CSV Header:', header)
print('Row count: ', len(rows))

>>>
CSV Header: ('Date', 'Make', 'Model', 'Year', 'Price')
Row count: 200
```

Unpacking with a starred expression makes it easy to process the first row—the header—separately from the rest of the iterator's contents. This is much clearer:

[Click here to view code image](#)

```
it = generate_csv()
header, *rows = it
print('CSV Header:', header)
print('Row count: ', len(rows))
```

```
>>>
CSV Header: ('Date', 'Make', 'Model', 'Year', 'Price')
Row count: 200
```

Keep in mind, however, that because a starred expression is always turned into a `list`, unpacking an iterator also risks the potential of using up all of the memory on your computer and causing your program to crash. So you should only use catch-all unpacking on iterators when you have good reason to believe that the result data will all fit in memory (see [Item 31: “Be Defensive When Iterating Over Arguments”](#) for another approach).

## Things to Remember

- ◆ Unpacking assignments may use a starred expression to catch all values that weren’t assigned to the other parts of the unpacking pattern into a `list`.
- ◆ Starred expressions may appear in any position, and they will always become a `list` containing the zero or more values they receive.
- ◆ When dividing a `list` into non-overlapping pieces, catch-all unpacking is much less error prone than slicing and indexing.

## Item 14: Sort by Complex Criteria Using the `key` Parameter

The `list` built-in type provides a `sort` method for ordering the items in a `list` instance based on a variety of criteria. By default, `sort` will order a `list`’s contents by the natural ascending order of the items. For example, here I sort a `list` of integers from smallest to largest:

```
numbers = [93, 86, 11, 68, 70]
numbers.sort()
print(numbers)
```

```
>>>
[11, 68, 70, 86, 93]
```

The `sort` method works for nearly all built-in types (strings, floats, etc.) that have a natural ordering to them. What does `sort` do with objects? For

example, here I define a class—including a `__repr__` method so instances are printable; see [Item 75: “Use repr Strings for Debugging Output”](#)—to represent various tools you may need to use on a construction site:

[Click here to view code image](#)

```
class Tool:
    def __init__(self, name, weight):
        self.name = name
        self.weight = weight

    def __repr__(self):
        return f'Tool({self.name!r}, {self.weight})'

tools = [
    Tool('level', 3.5),
    Tool('hammer', 1.25),
    Tool('screwdriver', 0.5),
    Tool('chisel', 0.25),
]
```

Sorting objects of this type doesn't work because the `sort` method tries to call comparison special methods that aren't defined by the class:

[Click here to view code image](#)

```
tools.sort()

>>>
Traceback ...
TypeError: '<' not supported between instances of 'Tool' and
'Tool'
```

If your class should have a natural ordering like integers do, then you can define the necessary special methods (see [Item 73: “Know How to Use `heapq` for Priority Queues”](#) for an example) to make `sort` work without extra parameters. But the more common case is that your objects may need to support multiple orderings, in which case defining a natural ordering really doesn't make sense.

Often there's an attribute on the object that you'd like to use for sorting. To support this use case, the `sort` method accepts a `key` parameter that's expected to be a function. The key function is passed a single argument,

which is an item from the `list` that is being sorted. The return value of the key function should be a comparable value (i.e., with a natural ordering) to use in place of an item for sorting purposes.

Here, I use the `lambda` keyword to define a function for the key parameter that enables me to sort the `list` of `Tool` objects alphabetically by their name:

[Click here to view code image](#)

```
print('Unsorted:', repr(tools))
tools.sort(key=lambda x: x.name)
print('\nSorted: ', tools)

>>>
Unsorted: [Tool('level',      3.5),
           Tool('hammer',    1.25),
           Tool('screwdriver', 0.5),
           Tool('chisel',     0.25)]

Sorted: [Tool('chisel',      0.25),
         Tool('hammer',    1.25),
         Tool('level',     3.5),
         Tool('screwdriver', 0.5)]
```

I can just as easily define another lambda function to sort by weight and pass it as the key parameter to the sort method:

[Click here to view code image](#)

```
tools.sort(key=lambda x: x.weight)
print('By weight:', tools)

>>>
By weight: [Tool('chisel',      0.25),
            Tool('screwdriver', 0.5),
            Tool('hammer',    1.25),
            Tool('level',     3.5)]
```

Within the lambda function passed as the key parameter you can access attributes of items as I've done here, index into items (for sequences, tuples, and dictionaries), or use any other valid expression.

For basic types like strings, you may even want to use the key function to do transformations on the values before sorting. For example, here I apply

the lower method to each item in a list of place names to ensure that they're in alphabetical order, ignoring any capitalization (since in the natural lexical ordering of strings, capital letters come before lowercase letters):

[Click here to view code image](#)

```
places = ['home', 'work', 'New York', 'Paris']
places.sort()
print('Case sensitive: ', places)
places.sort(key=lambda x: x.lower())
print('Case insensitive:', places)

>>>
Case sensitive: ['New York', 'Paris', 'home', 'work']
Case insensitive: ['home', 'New York', 'Paris', 'work']
```

Sometimes you may need to use multiple criteria for sorting. For example, say that I have a list of power tools and I want to sort them first by weight and then by name. How can I accomplish this?

```
power_tools = [
    Tool('drill', 4),
    Tool('circular saw', 5),
    Tool('jackhammer', 40),
    Tool('sander', 4),
]
```

The simplest solution in Python is to use the tuple type. Tuples are immutable sequences of arbitrary Python values. Tuples are comparable by default and have a natural ordering, meaning that they implement all of the special methods, such as `__lt__`, that are required by the `sort` method. Tuples implement these special method comparators by iterating over each position in the tuple and comparing the corresponding values one index at a time. Here, I show how this works when one tool is heavier than another:

[Click here to view code image](#)

```
saw = (5, 'circular saw')
jackhammer = (40, 'jackhammer')
assert not (jackhammer < saw) # Matches expectations
```

If the first position in the tuples being compared are equal—weight in this case—then the tuple comparison will move on to the second position, and so on:

[Click here to view code image](#)

```
drill = (4, 'drill')
sander = (4, 'sander')
assert drill[0] == sander[0] # Same weight
assert drill[1] < sander[1] # Alphabetically less
assert drill < sander        # Thus, drill comes first
```

You can take advantage of this tuple comparison behavior in order to sort the list of power tools first by weight and then by name. Here, I define a key function that returns a tuple containing the two attributes that I want to sort on in order of priority:

[Click here to view code image](#)

```
power_tools.sort(key=lambda x: (x.weight, x.name))
print(power_tools)

>>>
[Tool('drill', 4),
 Tool('sander', 4),
 Tool('circular saw', 5),
 Tool('jackhammer', 40)]
```

One limitation of having the key function return a tuple is that the direction of sorting for all criteria must be the same (either all in ascending order, or all in descending order). If I provide the reverse parameter to the sort method, it will affect both criteria in the tuple the same way (note how 'sander' now comes before 'drill' instead of after):

[Click here to view code image](#)

```
power_tools.sort(key=lambda x: (x.weight, x.name),
                  reverse=True) # Makes all criteria descending
print(power_tools)

>>>
[Tool('jackhammer', 40),
 Tool('circular saw', 5),
```

```
Tool('sander',      4),  
Tool('drill',       4)]
```

For numerical values it's possible to mix sorting directions by using the unary minus operator in the key function. This negates one of the values in the returned tuple, effectively reversing its sort order while leaving the others intact. Here, I use this approach to sort by weight descending, and then by name ascending (note how 'sander' now comes after 'drill' instead of before):

[Click here to view code image](#)

```
power_tools.sort(key=lambda x: (-x.weight, x.name))  
print(power_tools)
```

```
>>>  
[Tool('jackhammer', 40),  
  Tool('circular saw', 5),  
  Tool('drill', 4),  
  Tool('sander', 4)]
```

Unfortunately, unary negation isn't possible for all types. Here, I try to achieve the same outcome by using the reverse argument to sort by weight descending and then negating name to put it in ascending order:

[Click here to view code image](#)

```
power_tools.sort(key=lambda x: (x.weight, -x.name),  
                 reverse=True)
```

```
>>>  
Traceback ...  
TypeError: bad operand type for unary -: 'str'
```

For situations like this, Python provides a *stable* sorting algorithm. The sort method of the list type will preserve the order of the input list when the key function returns values that are equal to each other. This means that I can call sort multiple times on the same list to combine different criteria together. Here, I produce the same sort ordering of weight descending and name ascending as I did above but by using two separate calls to sort:

[Click here to view code image](#)



```
power_tools.sort(key=lambda x: x.name) # Name ascending
power_tools.sort(key=lambda x: x.weight, # Weight descending
                  reverse=True)
print(power_tools)
```

```
>>>
[Tool('jackhammer', 40),
 Tool('circular saw', 5),
 Tool('drill', 4),
 Tool('sander', 4)]
```

To understand why this works, note how the first call to sort puts the names in alphabetical order:

[Click here to view code image](#)

```
power_tools.sort(key=lambda x: x.name)
print(power_tools)
```

```
>>>
[Tool('circular saw', 5),
 Tool('drill', 4),
 Tool('jackhammer', 40),
 Tool('sander', 4)]
```

When the second sort call by weight descending is made, it sees that both 'sander' and 'drill' have a weight of 4. This causes the sort method to put both items into the final result list in the same order that they appeared in the original list, thus preserving their relative ordering by name ascending:

[Click here to view code image](#)

```
power_tools.sort(key=lambda x: x.weight,
                  reverse=True)
print(power_tools)
```

```
>>>
[Tool('jackhammer', 40),
 Tool('circular saw', 5),
 Tool('drill', 4),
 Tool('sander', 4)]
```

This same approach can be used to combine as many different types of sorting criteria as you'd like in any direction, respectively. You just need to

make sure that you execute the sorts in the opposite sequence of what you want the final `list` to contain. In this example, I wanted the sort order to be by weight descending and then by name ascending, so I had to do the name sort first, followed by the weight sort.

That said, the approach of having the key function return a tuple, and using unary negation to mix sort orders, is simpler to read and requires less code. I recommend only using multiple calls to sort if it's absolutely necessary.

## Things to Remember

- ◆ The `sort` method of the `list` type can be used to rearrange a list's contents by the natural ordering of built-in types like strings, integers, tuples, and so on.
- ◆ The `sort` method doesn't work for objects unless they define a natural ordering using special methods, which is uncommon.
- ◆ The `key` parameter of the `sort` method can be used to supply a helper function that returns the value to use for sorting in place of each item from the `list`.
- ◆ Returning a tuple from the key function allows you to combine multiple sorting criteria together. The unary minus operator can be used to reverse individual sort orders for types that allow it.
- ◆ For types that can't be negated, you can combine many sorting criteria together by calling the `sort` method multiple times using different key functions and reverse values, in the order of lowest rank sort call to highest rank sort call.

## Item 15: Be Cautious When Relying on `dict` Insertion Ordering

In Python 3.5 and before, iterating over a `dict` would return keys in arbitrary order. The order of iteration would not match the order in which the items were inserted. For example, here I create a dictionary mapping

animal names to their corresponding baby names and then print it out (see [Item 75: “Use repr Strings for Debugging Output”](#) for how this works):

```
# Python 3.5
baby_names = {
    'cat': 'kitten',
    'dog': 'puppy',
}
print(baby_names)

>>>
{'dog': 'puppy', 'cat': 'kitten'}
```

When I created the dictionary the keys were in the order 'cat', 'dog', but when I printed it the keys were in the reverse order 'dog', 'cat'. This behavior is surprising, makes it harder to reproduce test cases, increases the difficulty of debugging, and is especially confusing to newcomers to Python.

This happened because the dictionary type previously implemented its hash table algorithm with a combination of the `hash` built-in function and a random seed that was assigned when the Python interpreter started. Together, these behaviors caused dictionary orderings to not match insertion order and to randomly shuffle between program executions.

Starting with Python 3.6, and officially part of the Python specification in version 3.7, dictionaries will preserve insertion order. Now, this code will always print the dictionary in the same way it was originally created by the programmer:

```
baby_names = {
    'cat': 'kitten',
    'dog': 'puppy',
}
print(baby_names)

>>>
{'cat': 'kitten', 'dog': 'puppy'}
```

With Python 3.5 and earlier, all methods provided by `dict` that relied on iteration order, including `keys`, `values`, `items`, and `popitem`, would similarly demonstrate this random-looking behavior:

[Click here to view code image](#)

```
# Python 3.5
print(list(baby_names.keys()))
print(list(baby_names.values()))
print(list(baby_names.items()))
print(baby_names.popitem()) # Randomly chooses an item

>>>
['dog', 'cat']
['puppy', 'kitten']
[('dog', 'puppy'), ('cat', 'kitten')]
('dog', 'puppy')
```

These methods now provide consistent insertion ordering that you can rely on when you write your programs:

[Click here to view code image](#)

```
print(list(baby_names.keys()))
print(list(baby_names.values()))
print(list(baby_names.items()))
print(baby_names.popitem()) # Last item inserted

>>>
['cat', 'dog']
['kitten', 'puppy']
[('cat', 'kitten'), ('dog', 'puppy')]
('dog', 'puppy')
```

There are many repercussions of this change on other Python features that are dependent on the dict type and its specific implementation.

Keyword arguments to functions—including the `**kwargs` catch-all parameter; see [Item 23: “Provide Optional Behavior with Keyword Arguments”](#)—previously would come through in seemingly random order, which can make it harder to debug function calls:

[Click here to view code image](#)

```
# Python 3.5
def my_func(**kwargs):
    for key, value in kwargs.items():
        print('%s = %s' % (key, value))

my_func(goose='gosling', kangaroo='joey')
```

```
>>>
kangaroo = joey
goose = gosling
```

Now, the order of keyword arguments is always preserved to match how the programmer originally called the function:

[Click here to view code image](#)

```
def my_func(**kwargs):
    for key, value in kwargs.items():
        print(f'{key} = {value}')

my_func(goose='gosling', kangaroo='joey')
```

```
>>>
goose = gosling
kangaroo = joey
```

Classes also use the dict type for their instance dictionaries. In previous versions of Python, object fields would show the randomizing behavior:

[Click here to view code image](#)

```
# Python 3.5
class MyClass:
    def __init__(self):
        self.alligator = 'hatchling'
        self.elephant = 'calf'
a = MyClass()
for key, value in a.__dict__.items():
    print('%s = %s' % (key, value))
```

```
>>>
elephant = calf
alligator = hatchling
```

Again, you can now assume that the order of assignment for these instance fields will be reflected in `__dict__`:

[Click here to view code image](#)

```
class MyClass:
    def __init__(self):
        self.alligator = 'hatchling'
        self.elephant = 'calf'
```

```
a = MyClass()
for key, value in a.__dict__.items():
    print(f'{key} = {value}')

>>>
alligator = hatchling
elephant = calf
```

The way that dictionaries preserve insertion ordering is now part of the Python language specification. For the language features above, you can rely on this behavior and even make it part of the APIs you design for your classes and functions.

## Note

For a long time the `collections` built-in module has had an `OrderedDict` class that preserves insertion ordering. Although this class’s behavior is similar to that of the standard `dict` type (since Python 3.7), the performance characteristics of `OrderedDict` are quite different. If you need to handle a high rate of key insertions and `popitem` calls (e.g., to implement a least-recently-used cache), `OrderedDict` may be a better fit than the standard Python `dict` type (see [Item 70: “Profile Before Optimizing”](#) on how to make sure you need this).

However, you shouldn’t always assume that insertion ordering behavior will be present when you’re handling dictionaries. Python makes it easy for programmers to define their own custom container types that emulate the standard *protocols* matching `list`, `dict`, and other types (see [Item 43: “Inherit from `collections.abc` for Custom Container Types”](#)). Python is not statically typed, so most code relies on *duck typing*—where an object’s behavior is its de facto type—instead of rigid class hierarchies. This can result in surprising gotchas.

For example, say that I’m writing a program to show the results of a contest for the cutest baby animal. Here, I start with a dictionary containing the total vote count for each one:

```
votes = {
    'otter': 1281,
    'polar bear': 587,
```

```
        'fox': 863,  
    }
```

I define a function to process this voting data and save the rank of each animal name into a provided empty dictionary. In this case, the dictionary could be the data model that powers a UI element:

[Click here to view code image](#)

```
def populate_ranks(votes, ranks):  
    names = list(votes.keys())  
    names.sort(key=votes.get, reverse=True)  
    for i, name in enumerate(names, 1):  
        ranks[name] = i
```

I also need a function that will tell me which animal won the contest. This function works by assuming that `populate_ranks` will assign the contents of the `ranks` dictionary in ascending order, meaning that the first key must be the winner:

```
def get_winner(ranks):  
    return next(iter(ranks))
```

Here, I can confirm that these functions work as designed and deliver the result that I expected:

[Click here to view code image](#)

```
ranks = {}  
populate_ranks(votes, ranks)  
print(ranks)  
winner = get_winner(ranks)  
print(winner)  
  
>>>  
{'otter': 1, 'fox': 2, 'polar bear': 3}  
otter
```

Now, imagine that the requirements of this program have changed. The UI element that shows the results should be in alphabetical order instead of rank order. To accomplish this, I can use the `collections.abc` built-in module to define a new dictionary-like class that iterates its contents in alphabetical order:

[Click here to view code image](#)

```
from collections.abc import MutableMapping

class SortedDict(MutableMapping):
    def __init__(self):
        self.data = {}

    def __getitem__(self, key):
        return self.data[key]
    def __setitem__(self, key, value):
        self.data[key] = value

    def __delitem__(self, key):
        del self.data[key]
    def __iter__(self):
        keys = list(self.data.keys())
        keys.sort()
        for key in keys:
            yield key
    def __len__(self):
        return len(self.data)
```

I can use a SortedDict instance in place of a standard dict with the functions from before and no errors will be raised since this class conforms to the protocol of a standard dictionary. However, the result is incorrect:

[Click here to view code image](#)

```
sorted_ranks = SortedDict()
populate_ranks(votes, sorted_ranks)
print(sorted_ranks.data)
winner = get_winner(sorted_ranks)
print(winner)

>>>
{'otter': 1, 'fox': 2, 'polar bear': 3}
fox
```

The problem here is that the implementation of `get_winner` assumes that the dictionary's iteration is in insertion order to match `populate_ranks`. This code is using `SortedDict` instead of `dict`, so that assumption is no longer true. Thus, the value returned for the winner is `'fox'`, which is alphabetically first.



There are three ways to mitigate this problem. First, I can reimplement the `get_winner` function to no longer assume that the ranks dictionary has a specific iteration order. This is the most conservative and robust solution:

[Click here to view code image](#)

```
def get_winner(ranks):
    for name, rank in ranks.items():
        if rank == 1:
            return name

winner = get_winner(sorted_ranks)
print(winner)

>>>
otter
```

The second approach is to add an explicit check to the top of the function to ensure that the type of ranks matches my expectations, and to raise an exception if not. This solution likely has better runtime performance than the more conservative approach:

[Click here to view code image](#)

```
def get_winner(ranks):
    if not isinstance(ranks, dict):
        raise TypeError('must provide a dict instance')
    return next(iter(ranks))

get_winner(sorted_ranks)

>>>
Traceback ...
TypeError: must provide a dict instance
```

The third alternative is to use type annotations to enforce that the value passed to `get_winner` is a dict instance and not a `MutableMapping` with dictionary-like behavior (see [Item 90: “Consider Static Analysis via typing to Obviate Bugs”](#)). Here, I run the `mypy` tool in strict mode on an annotated version of the code above:

[Click here to view code image](#)

```

from typing import Dict, MutableMapping

def populate_ranks(votes: Dict[str, int],
                  ranks: Dict[str, int]) -> None:
    names = list(votes.keys())
    names.sort(key=votes.get, reverse=True)
    for i, name in enumerate(names, 1):
        ranks[name] = i

def get_winner(ranks: Dict[str, int]) -> str:
    return next(iter(ranks))
class SortedDict(MutableMapping[str, int]):
    ...

votes = {
    'otter': 1281,
    'polar bear': 587,
    'fox': 863,
}
sorted_ranks = SortedDict()
populate_ranks(votes, sorted_ranks)
print(sorted_ranks.data)
winner = get_winner(sorted_ranks)
print(winner)

```

[Click here to view code image](#)

```

$ python3 -m mypy --strict example.py
.../example.py:48: error: Argument 2 to "populate_ranks" has
➔ incompatible type "SortedDict"; expected "Dict[str, int]"
.../example.py:50: error: Argument 1 to "get_winner" has
➔ incompatible type "SortedDict"; expected "Dict[str, int]"

```

This correctly detects the mismatch between the dict and MutableMapping types and flags the incorrect usage as an error. This solution provides the best mix of static type safety and runtime performance.

## Things to Remember

- ◆ Since Python 3.7, you can rely on the fact that iterating a dict instance's contents will occur in the same order in which the keys were initially added.
- ◆ Python makes it easy to define objects that act like dictionaries but that aren't dict instances. For these types, you can't assume that insertion

ordering will be preserved.

- ♦ There are three ways to be careful about dictionary-like classes: Write code that doesn't rely on insertion ordering, explicitly check for the `dict` type at runtime, or require `dict` values using type annotations and static analysis.

## Item 16: Prefer `get` Over `in` and `KeyError` to Handle Missing Dictionary Keys

The three fundamental operations for interacting with dictionaries are accessing, assigning, and deleting keys and their associated values. The contents of dictionaries are dynamic, and thus it's entirely possible—even likely—that when you try to access or delete a key, it won't already be present.

For example, say that I'm trying to determine people's favorite type of bread to devise the menu for a sandwich shop. Here, I define a dictionary of counters with the current votes for each style:

```
counters = {  
    'pumpernickel': 2,  
    'sourdough': 1,  
}
```

To increment the counter for a new vote, I need to see if the key exists, insert the key with a default counter value of zero if it's missing, and then increment the counter's value. This requires accessing the key two times and assigning it once. Here, I accomplish this task using an `if` statement with an `in` expression that returns `True` when the key is present:

```
key = 'wheat'  
  
if key in counters:  
    count = counters[key]  
else:  
    count = 0  
  
counters[key] = count + 1
```

Another way to accomplish the same behavior is by relying on how dictionaries raise a `KeyError` exception when you try to get the value for a key that doesn't exist. This approach is more efficient because it requires only one access and one assignment:

```
try:
    count = counters[key]
except KeyError:
    count = 0

counters[key] = count + 1
```

This flow of fetching a key that exists or returning a default value is so common that the `dict` built-in type provides the `get` method to accomplish this task. The second parameter to `get` is the default value to return in the case that the key—the first parameter—isn't present. This also requires only one access and one assignment, but it's much shorter than the `KeyError` example:

```
count = counters.get(key, 0)
counters[key] = count + 1
```

It's possible to shorten the `in` expression and `KeyError` approaches in various ways, but all of these alternatives suffer from requiring code duplication for the assignments, which makes them less readable and worth avoiding:

```
if key not in counters:
    counters[key] = 0
counters[key] += 1

if key in counters:
    counters[key] += 1
else:
    counters[key] = 1

try:
    counters[key] += 1
except KeyError:
    counters[key] = 1
```

Thus, for a dictionary with simple types, using the `get` method is the shortest and clearest option.

## Note

If you're maintaining dictionaries of counters like this, it's worth considering the `Counter` class from the `collections` built-in module, which provides most of the facilities you are likely to need.

What if the values of the dictionary are a more complex type, like a `list`? For example, say that instead of only counting votes, I also want to know who voted for each type of bread. Here, I do this by associating a `list` of names with each key:

```
votes = {
    'baguette': ['Bob', 'Alice'],
    'ciabatta': ['Coco', 'Deb'],
}
key = 'brioche'
who = 'Elmer'

if key in votes:
    names = votes[key]
else:
    votes[key] = names = []

names.append(who)
print(votes)

>>>
{'baguette': ['Bob', 'Alice'],
 'ciabatta': ['Coco', 'Deb'],
 'brioche': ['Elmer']}
```

Relying on the `in` expression requires two accesses if the key is present, or one access and one assignment if the key is missing. This example is different from the counters example above because the value for each key can be assigned blindly to the default value of an empty `list` if the key doesn't already exist. The triple assignment statement (`votes[key] = names = []`) populates the key in one line instead of two. Once the default value has been inserted into the dictionary, I don't need to assign it again because the `list` is modified by reference in the later call to `append`.

It's also possible to rely on the `KeyError` exception being raised when the dictionary value is a `list`. This approach requires one key access if the key

is present, or one key access and one assignment if it's missing, which makes it more efficient than the `in` condition:

```
try:
    names = votes[key]
except KeyError:
    votes[key] = names = []

names.append(who)
```

Similarly, you can use the `get` method to fetch a `list` value when the key is present, or do one fetch and one assignment if the key isn't present:

```
names = votes.get(key)
if names is None:
    votes[key] = names = []

names.append(who)
```

The approach that involves using `get` to fetch `list` values can further be shortened by one line if you use an assignment expression (introduced in Python 3.8; see [Item 10: “Prevent Repetition with Assignment Expressions”](#)) in the `if` statement, which improves readability:

[Click here to view code image](#)

```
if (names := votes.get(key)) is None:
    votes[key] = names = []

names.append(who)
```

The `dict` type also provides the `setdefault` method to help shorten this pattern even further. `setdefault` tries to fetch the value of a key in the dictionary. If the key isn't present, the method assigns that key to the default value provided. And then the method returns the value for that key: either the originally present value or the newly inserted default value. Here, I use `setdefault` to implement the same logic as in the `get` example above:

```
names = votes.setdefault(key, [])
names.append(who)
```

This works as expected, and it is shorter than using `get` with an assignment expression. However, the readability of this approach isn't ideal. The

method name `setdefault` doesn't make its purpose immediately obvious. Why is it set when what it's doing is getting a value? Why not call it `get_or_set`? I'm arguing about the color of the bike shed here, but the point is that if you were a new reader of the code and not completely familiar with Python, you might have trouble understanding what this code is trying to accomplish because `setdefault` isn't self-explanatory.

There's also one important gotcha: The default value passed to `setdefault` is assigned directly into the dictionary when the key is missing instead of being copied. Here, I demonstrate the effect of this when the value is a list:

```
data = {}
key = 'foo'
value = []
data.setdefault(key, value)
print('Before:', data)
value.append('hello')
print('After: ', data)
```

```
>>>
Before: {'foo': []}
After: {'foo': ['hello']}
```

This means that I need to make sure that I'm always constructing a new default value for each key I access with `setdefault`. This leads to a significant performance overhead in this example because I have to allocate a list instance for each call. If I reuse an object for the default value—which I might try to do to increase efficiency or readability—I might introduce strange behavior and bugs (see [Item 24: “Use None and Docstrings to Specify Dynamic Default Arguments”](#) for another example of this problem).

Going back to the earlier example that used counters for dictionary values instead of lists of who voted: Why not also use the `setdefault` method in that case? Here, I reimplement the same example using this approach:

```
count = counters.setdefault(key, 0)
counters[key] = count + 1
```

The problem here is that the call to `setdefault` is superfluous. You always need to assign the key in the dictionary to a new value after you increment the counter, so the extra assignment done by `setdefault` is unnecessary. The earlier approach of using `get` for counter updates requires only one access and one assignment, whereas using `setdefault` requires one access and two assignments.

There are only a few circumstances in which using `setdefault` is the shortest way to handle missing dictionary keys, such as when the default values are cheap to construct, mutable, and there's no potential for raising exceptions (e.g., `list` instances). In these very specific cases, it may seem worth accepting the confusing method name `setdefault` instead of having to write more characters and lines to use `get`. However, often what you really should do in these situations is to use `defaultdict` instead (see [Item 17: “Prefer `defaultdict` Over `setdefault` to Handle Missing Items in Internal State](#)”).

## Things to Remember

- ◆ There are four common ways to detect and handle missing keys in dictionaries: using `in` expressions, `KeyError` exceptions, the `get` method, and the `setdefault` method.
- ◆ The `get` method is best for dictionaries that contain basic types like counters, and it is preferable along with assignment expressions when creating dictionary values has a high cost or may raise exceptions.
- ◆ When the `setdefault` method of `dict` seems like the best fit for your problem, you should consider using `defaultdict` instead.

## Item 17: Prefer `defaultdict` Over `setdefault` to Handle Missing Items in Internal State

When working with a dictionary that you didn't create, there are a variety of ways to handle missing keys (see [Item 16: “Prefer `get` Over `in` and `KeyError` to Handle Missing Dictionary Keys](#)”). Although using the `get`



method is a better approach than using `in` expressions and `KeyError` exceptions, for some use cases `setdefault` appears to be the shortest option.

For example, say that I want to keep track of the cities I've visited in countries around the world. Here, I do this by using a dictionary that maps country names to a set instance containing corresponding city names:

[Click here to view code image](#)

```
visits = {
    'Mexico': {'Tulum', 'Puerto Vallarta'},
    'Japan': {'Hakone'},
}
```

I can use the `setdefault` method to add new cities to the sets, whether the country name is already present in the dictionary or not. This approach is much shorter than achieving the same behavior with the `get` method and an assignment expression (which is available as of Python 3.8):

[Click here to view code image](#)

```
visits.setdefault('France', set()).add('Arles') # Short

if (japan := visits.get('Japan')) is None:      # Long
    visits['Japan'] = japan = set()
japan.add('Kyoto')

print(visits)

>>>
{'Mexico': {'Tulum', 'Puerto Vallarta'},
 'Japan': {'Kyoto', 'Hakone'},
 'France': {'Arles'}}
```

What about the situation when you *do* control creation of the dictionary being accessed? This is generally the case when you're using a dictionary instance to keep track of the internal state of a class, for example. Here, I wrap the example above in a class with helper methods to access the dynamic inner state stored in a dictionary:

[Click here to view code image](#)

```
class Visits:
    def __init__(self):
```

```

self.data = {}

def add(self, country, city):
    city_set = self.data.setdefault(country, set())
    city_set.add(city)

```

This new class hides the complexity of calling `setdefault` correctly, and it provides a nicer interface for the programmer:

[Click here to view code image](#)

```

visits = Visits()
visits.add('Russia', 'Yekaterinburg')
visits.add('Tanzania', 'Zanzibar')
print(visits.data)

>>>
{'Russia': {'Yekaterinburg'}, 'Tanzania': {'Zanzibar'}}

```

However, the implementation of the `visits.add` method still isn't ideal. The `setdefault` method is still confusingly named, which makes it more difficult for a new reader of the code to immediately understand what's happening. And the implementation isn't efficient because it constructs a new set instance on every call, regardless of whether the given country was already present in the data dictionary.

Luckily, the `defaultdict` class from the `collections` built-in module simplifies this common use case by automatically storing a default value when a key doesn't exist. All you have to do is provide a function that will return the default value to use each time a key is missing (an example of [Item 38: “Accept Functions Instead of Classes for Simple Interfaces”](#)).

Here, I rewrite the `visits` class to use `defaultdict`:

[Click here to view code image](#)

```

from collections import defaultdict

class Visits:
    def __init__(self):
        self.data = defaultdict(set)

    def add(self, country, city):
        self.data[country].add(city)

```

```
visits = Visits()
visits.add('England', 'Bath')
visits.add('England', 'London')
print(visits.data)

>>>
defaultdict(<class 'set'>, {'England': {'London', 'Bath'}})
```

Now, the implementation of `add` is short and simple. The code can assume that accessing any key in the data dictionary will always result in an existing set instance. No superfluous set instances will be allocated, which could be costly if the `add` method is called a large number of times.

Using `defaultdict` is much better than using `setdefault` for this type of situation (see [Item 37: “Compose Classes Instead of Nesting Many Levels of Built-in Types”](#) for another example). There are still cases in which `defaultdict` will fall short of solving your problems, but there are even more tools available in Python to work around those limitations (see [Item 18: “Know How to Construct Key-Dependent Default Values with `\_\_missing\_\_`,”](#) [Item 43: “Inherit from `collections.abc` for Custom Container Types,”](#) and the `collections.Counter` built-in class).

## Things to Remember

- ◆ If you’re creating a dictionary to manage an arbitrary set of potential keys, then you should prefer using a `defaultdict` instance from the `collections` built-in module if it suits your problem.
- ◆ If a dictionary of arbitrary keys is passed to you, and you don’t control its creation, then you should prefer the `get` method to access its items. However, it’s worth considering using the `setdefault` method for the few situations in which it leads to shorter code.

## Item 18: Know How to Construct Key-Dependent Default Values with `__missing__`

The built-in `dict` type’s `setdefault` method results in shorter code when handling missing keys in some specific circumstances (see [Item 16: “Prefer](#)

[get Over in and KeyError to Handle Missing Dictionary Keys](#)” for examples). For many of those situations, the better tool for the job is the `defaultdict` type from the `collections` built-in module (see [Item 17: “Prefer defaultdict Over setdefault to Handle Missing Items in Internal State”](#) for why). However, there are times when neither `setdefault` nor `defaultdict` is the right fit.

For example, say that I’m writing a program to manage social network profile pictures on the filesystem. I need a dictionary to map profile picture pathnames to open file handles so I can read and write those images as needed. Here, I do this by using a normal `dict` instance and checking for the presence of keys using the `get` method and an assignment expression (introduced in Python 3.8; see [Item 10: “Prevent Repetition with Assignment Expressions”](#)):

[Click here to view code image](#)

```
pictures = {}
path = 'profile_1234.png'

if (handle := pictures.get(path)) is None:
    try:
        handle = open(path, 'a+b')
    except OSError:
        print(f'Failed to open path {path}')
        raise
    else:
        pictures[path] = handle

handle.seek(0)
image_data = handle.read()
```

When the file handle already exists in the dictionary, this code makes only a single dictionary access. In the case that the file handle doesn’t exist, the dictionary is accessed once by `get`, and then it is assigned in the `else` clause of the `try/except` block. (This approach also works with `finally`; see [Item 65: “Take Advantage of Each Block in try/except/else/finally.”](#)) The call to the `read` method stands clearly separate from the code that calls `open` and handles exceptions.

Although it's possible to use the `in` expression or `KeyError` approaches to implement this same logic, those options require more dictionary accesses and levels of nesting. Given that these other options work, you might also assume that the `setdefault` method would work, too:

[Click here to view code image](#)

```
try:
    handle = pictures.setdefault(path, open(path, 'a+b'))
except OSError:
    print(f'Failed to open path {path}')
    raise
else:
    handle.seek(0)
    image_data = handle.read()
```

This code has many problems. The `open` built-in function to create the file handle is always called, even when the path is already present in the dictionary. This results in an additional file handle that may conflict with existing open handles in the same program. Exceptions may be raised by the `open` call and need to be handled, but it may not be possible to differentiate them from exceptions that may be raised by the `setdefault` call on the same line (which is possible for other dictionary-like implementations; see [Item 43: “Inherit from `collections.abc` for Custom Container Types](#)”).

If you're trying to manage internal state, another assumption you might make is that a `defaultdict` could be used for keeping track of these profile pictures. Here, I attempt to implement the same logic as before but now using a helper function and the `defaultdict` class:

[Click here to view code image](#)

```
from collections import defaultdict

def open_picture(profile_path):
    try:
        return open(profile_path, 'a+b')
    except OSError:
        print(f'Failed to open path {profile_path}')
        raise
```

```

pictures = defaultdict(open_picture)
handle = pictures[path]
handle.seek(0)
image_data = handle.read()

>>>
Traceback ...
TypeError: open_picture() missing 1 required positional
argument: 'profile_path'

```

The problem is that `defaultdict` expects that the function passed to its constructor doesn't require any arguments. This means that the helper function that `defaultdict` calls doesn't know which specific key is being accessed, which eliminates my ability to call `open`. In this situation, both `setdefault` and `defaultdict` fall short of what I need.

Fortunately, this situation is common enough that Python has another built-in solution. You can subclass the `dict` type and implement the `__missing__` special method to add custom logic for handling missing keys. Here, I do this by defining a new class that takes advantage of the same `open_picture` helper method defined above:

```

class Pictures(dict):
    def __missing__(self, key):
        value = open_picture(key)
        self[key] = value
        return value

pictures = Pictures()
handle = pictures[path]
handle.seek(0)
image_data = handle.read()

```

When the `pictures[path]` dictionary access finds that the `path` key isn't present in the dictionary, the `__missing__` method is called. This method must create the new default value for the key, insert it into the dictionary, and return it to the caller. Subsequent accesses of the same `path` will not call `__missing__` since the corresponding item is already present (similar to the behavior of `__getattr__`; see [Item 47: “Use `\_\_getattr\_\_`, `\_\_getattribute\_\_`, and `\_\_setattr\_\_` for Lazy Attributes](#)”).

## Things to Remember

- ✦ The `setdefault` method of `dict` is a bad fit when creating the default value has high computational cost or may raise exceptions.
- ✦ The function passed to `defaultdict` must not require any arguments, which makes it impossible to have the default value depend on the key being accessed.
- ✦ You can define your own `dict` subclass with a `__missing__` method in order to construct default values that must know which key was being accessed.

## 3. Functions

The first organizational tool programmers use in Python is the *function*. As in other programming languages, functions enable you to break large programs into smaller, simpler pieces with names to represent their intent. They improve readability and make code more approachable. They allow for reuse and refactoring.

Functions in Python have a variety of extra features that make a programmer's life easier. Some are similar to capabilities in other programming languages, but many are unique to Python. These extras can make a function's purpose more obvious. They can eliminate noise and clarify the intention of callers. They can significantly reduce subtle bugs that are difficult to find.

### Item 19: Never Unpack More Than Three Variables When Functions Return Multiple Values

One effect of the unpacking syntax (see [Item 6: “Prefer Multiple Assignment Unpacking Over Indexing”](#)) is that it allows Python functions to seemingly return more than one value. For example, say that I'm trying to determine various statistics for a population of alligators. Given a `list` of `lengths`, I need to calculate the minimum and maximum lengths in the population. Here, I do this in a single function that appears to return two values:

[Click here to view code image](#)

```
def get_stats(numbers):  
    minimum = min(numbers)  
    maximum = max(numbers)  
    return minimum, maximum  
  
lengths = [63, 73, 72, 60, 67, 66, 71, 61, 72, 70]  
  
minimum, maximum = get_stats(lengths) # Two return values  
  
print(f'Min: {minimum}, Max: {maximum}')
```



```
>>>
Min: 60, Max: 73
```

The way this works is that multiple values are returned together in a two-item tuple. The calling code then unpacks the returned tuple by assigning two variables. Here, I use an even simpler example to show how an unpacking statement and multiple-return function work the same way:

```
first, second = 1, 2
assert first == 1
assert second == 2

def my_function():
    return 1, 2

first, second = my_function()
assert first == 1
assert second == 2
```

Multiple return values can also be received by starred expressions for catch-all unpacking (see [Item 13: “Prefer Catch-All Unpacking Over Slicing”](#)). For example, say I need another function that calculates how big each alligator is relative to the population average. This function returns a list of ratios, but I can receive the longest and shortest items individually by using a starred expression for the middle portion of the list:

[Click here to view code image](#)

```
def get_avg_ratio(numbers):
    average = sum(numbers) / len(numbers)
    scaled = [x / average for x in numbers]
    scaled.sort(reverse=True)
    return scaled

longest, *middle, shortest = get_avg_ratio(lengths)

print(f'Longest: {longest:>4.0%}')
print(f'Shortest: {shortest:>4.0%}')

>>>
Longest: 108%
Shortest: 89%
```

Now, imagine that the program's requirements change, and I need to also determine the average length, median length, and total population size of the alligators. I can do this by expanding the `get_stats` function to also calculate these statistics and return them in the result tuple that is unpacked by the caller:

[Click here to view code image](#)

```
def get_stats(numbers):
    minimum = min(numbers)
    maximum = max(numbers)
    count = len(numbers)
    average = sum(numbers) / count

    sorted_numbers = sorted(numbers)
    middle = count // 2
    if count % 2 == 0:
        lower = sorted_numbers[middle - 1]
        upper = sorted_numbers[middle]
        median = (lower + upper) / 2
    else:
        median = sorted_numbers[middle]

    return minimum, maximum, average, median, count

minimum, maximum, average, median, count = get_stats(lengths)

print(f'Min: {minimum}, Max: {maximum}')
print(f'Average: {average}, Median: {median}, Count {count}')

>>>
Min: 60, Max: 73
Average: 67.5, Median: 68.5, Count 10
```

There are two problems with this code. First, all the return values are numeric, so it is all too easy to reorder them accidentally (e.g., swapping average and median), which can cause bugs that are hard to spot later. Using a large number of return values is extremely error prone:

[Click here to view code image](#)

```
# Correct:
minimum, maximum, average, median, count = get_stats(lengths)
```

```
# Oops! Median and average swapped:
```

```
minimum, maximum, median, average, count = get_stats(lengths)
```

Second, the line that calls the function and unpacks the values is long, and it likely will need to be wrapped in one of a variety of ways (due to PEP8 style; see [Item 2: “Follow the PEP 8 Style Guide”](#)), which hurts readability:

[Click here to view code image](#)

```
minimum, maximum, average, median, count = get_stats(  
    lengths)
```

```
minimum, maximum, average, median, count = \  
    get_stats(lengths)
```

```
(minimum, maximum, average,  
 median, count) = get_stats(lengths)
```

```
(minimum, maximum, average, median, count  
 ) = get_stats(lengths)
```

To avoid these problems, you should never use more than three variables when unpacking the multiple return values from a function. These could be individual values from a three-tuple, two variables and one catch-all starred expression, or anything shorter. If you need to unpack more return values than that, you’re better off defining a lightweight class or `namedtuple` (see [Item 37: “Compose Classes Instead of Nesting Many Levels of Built-in Types”](#)) and having your function return an instance of that instead.

## Things to Remember

- ◆ You can have functions return multiple values by putting them in a tuple and having the caller take advantage of Python’s unpacking syntax.
- ◆ Multiple return values from a function can also be unpacked by catch-all starred expressions.
- ◆ Unpacking into four or more variables is error prone and should be avoided; instead, return a small class or `namedtuple` instance.

## Item 20: Prefer Raising Exceptions to Returning `None`

When writing utility functions, there's a draw for Python programmers to give special meaning to the return value of `None`. It seems to make sense in some cases. For example, say I want a helper function that divides one number by another. In the case of dividing by zero, returning `None` seems natural because the result is undefined:

```
def careful_divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        return None
```

Code using this function can interpret the return value accordingly:

```
x, y = 1, 0
result = careful_divide(x, y)
if result is None:
    print('Invalid inputs')
```

What happens with the `careful_divide` function when the numerator is zero? If the denominator is not zero, the function returns zero. The problem is that a zero return value can cause issues when you evaluate the result in a condition like an `if` statement. You might accidentally look for any `False`-equivalent value to indicate errors instead of only looking for `None` (see [Item 5: “Write Helper Functions Instead of Complex Expressions”](#) for a similar situation):

[Click here to view code image](#)

```
x, y = 0, 5
result = careful_divide(x, y)
if not result:
    print('Invalid inputs') # This runs! But shouldn't

>>>
Invalid inputs
```

This misinterpretation of a `False`-equivalent return value is a common mistake in Python code when `None` has special meaning. This is why

returning `None` from a function like `careful_divide` is error prone. There are two ways to reduce the chance of such errors.

The first way is to split the return value into a two-tuple (see [Item 19: “Never Unpack More Than Three Variables When Functions Return Multiple Values”](#) for background). The first part of the tuple indicates that the operation was a success or failure. The second part is the actual result that was computed:

```
def careful_divide(a, b):
    try:
        return True, a / b
    except ZeroDivisionError:
        return False, None
```

Callers of this function have to unpack the tuple. That forces them to consider the status part of the tuple instead of just looking at the result of division:

[Click here to view code image](#)

```
success, result = careful_divide(x, y)
if not success:
    print('Invalid inputs')
```

The problem is that callers can easily ignore the first part of the tuple (using the underscore variable name, a Python convention for unused variables). The resulting code doesn't look wrong at first glance, but this can be just as error prone as returning `None`:

```
_, result = careful_divide(x, y)
if not result:
    print('Invalid inputs')
```

The second, better way to reduce these errors is to never return `None` for special cases. Instead, raise an `Exception` up to the caller and have the caller deal with it. Here, I turn a `ZeroDivisionError` into a `ValueError` to indicate to the caller that the input values are bad (see [Item 87: “Define a Root Exception to Insulate Callers from APIs”](#) on when you should use `Exception` subclasses):

[Click here to view code image](#)

```
def careful_divide(a, b):
    try:
        return a / b
    except ZeroDivisionError as e:
        raise ValueError('Invalid inputs')
```

The caller no longer requires a condition on the return value of the function. Instead, it can assume that the return value is always valid and use the results immediately in the `else` block after `try` (see [Item 65: “Take Advantage of Each Block in try/except/else/finally”](#) for details):

[Click here to view code image](#)

```
x, y = 5, 2
try:
    result = careful_divide(x, y)
except ValueError:
    print('Invalid inputs')
else:
    print('Result is %.1f' % result)
```

```
>>>
Result is 2.5
```

This approach can be extended to code using type annotations (see [Item 90: “Consider Static Analysis via typing to Obviate Bugs”](#) for background). You can specify that a function’s return value will always be a `float` and thus will never be `None`. However, Python’s gradual typing purposefully doesn’t provide a way to indicate when exceptions are part of a function’s interface (also known as *checked exceptions*). Instead, you have to document the exception-raising behavior and expect callers to rely on that in order to know which Exceptions they should plan to catch (see [Item 84: “Write Docstrings for Every Function, Class, and Module”](#)).

Pulling it all together, here’s what this function should look like when using type annotations and docstrings:

[Click here to view code image](#)

```
def careful_divide(a: float, b: float) -> float:
    """Divides a by b.

    Raises:
```

```

        ValueError: When the inputs cannot be divided.
    """

    try:
        return a / b
    except ZeroDivisionError as e:
        raise ValueError('Invalid inputs')

```

Now the inputs, outputs, and exceptional behavior is clear, and the chance of a caller doing the wrong thing is extremely low.

## Things to Remember

- ◆ Functions that return `None` to indicate special meaning are error prone because `None` and other values (e.g., zero, the empty string) all evaluate to `False` in conditional expressions.
- ◆ Raise exceptions to indicate special situations instead of returning `None`. Expect the calling code to handle exceptions properly when they're documented.
- ◆ Type annotations can be used to make it clear that a function will never return the value `None`, even in special situations.

## Item 21: Know How Closures Interact with Variable Scope

Say that I want to sort a `list` of numbers but prioritize one group of numbers to come first. This pattern is useful when you're rendering a user interface and want important messages or exceptional events to be displayed before everything else.

A common way to do this is to pass a helper function as the key argument to a `list`'s `sort` method (see [Item 14: “Sort by Complex Criteria Using the `key` Parameter”](#) for details). The helper's return value will be used as the value for sorting each item in the `list`. The helper can check whether the given item is in the important group and can vary the sorting value accordingly:

```

def sort_priority(values, group):
    def helper(x):

```

```
    if x in group:
        return (0, x)
    return (1, x)
values.sort(key=helper)
```

This function works for simple inputs:

```
numbers = [8, 3, 1, 2, 5, 4, 7, 6]
group = {2, 3, 5, 7}
sort_priority(numbers, group)
print(numbers)
```

```
>>>
```

```
[2, 3, 5, 7, 1, 4, 6, 8]
```

There are three reasons this function operates as expected:

- Python supports *closures*—that is, functions that refer to variables from the scope in which they were defined. This is why the `helper` function is able to access the `group` argument for `sort_priority`.
- Functions are *first-class* objects in Python, which means you can refer to them directly, assign them to variables, pass them as arguments to other functions, compare them in expressions and `if` statements, and so on. This is how the `sort` method can accept a closure function as the `key` argument.
- Python has specific rules for comparing sequences (including tuples). It first compares items at index zero; then, if those are equal, it compares items at index one; if they are still equal, it compares items at index two, and so on. This is why the return value from the `helper` closure causes the sort order to have two distinct groups.

It'd be nice if this function returned whether higher-priority items were seen at all so the user interface code can act accordingly. Adding such behavior seems straightforward. There's already a closure function for deciding which group each number is in. Why not also use the closure to flip a flag when high-priority items are seen? Then, the function can return the flag value after it's been modified by the closure.

Here, I try to do that in a seemingly obvious way:



[Click here to view code image](#)

```
def sort_priority2(numbers, group):
    found = False
    def helper(x):
        if x in group:
            found = True # Seems simple
            return (0, x)
        return (1, x)
    numbers.sort(key=helper)
    return found
```

I can run the function on the same inputs as before:

[Click here to view code image](#)

```
found = sort_priority2(numbers, group)
print('Found:', found)
print(numbers)
```

```
>>>
Found: False
[2, 3, 5, 7, 1, 4, 6, 8]
```

The sorted results are correct, which means items from group were definitely found in numbers. Yet the found result returned by the function is False when it should be True. How could this happen?

When you reference a variable in an expression, the Python interpreter traverses the scope to resolve the reference in this order:

1. The current function's scope.
2. Any enclosing scopes (such as other containing functions).
3. The scope of the module that contains the code (also called the *global scope*).
4. The built-in scope (that contains functions like len and str).

If none of these places has defined a variable with the referenced name, then a NameError exception is raised:

[Click here to view code image](#)

```
foo = does_not_exist * 5
```

```
>>>
```

```
Traceback ...
```

```
NameError: name 'does_not_exist' is not defined
```

Assigning a value to a variable works differently. If the variable is already defined in the current scope, it will just take on the new value. If the variable doesn't exist in the current scope, Python treats the assignment as a variable definition. Critically, the scope of the newly defined variable is the function that contains the assignment.

This assignment behavior explains the wrong return value of the `sort_priority2` function. The `found` variable is assigned to `True` in the helper closure. The closure's assignment is treated as a new variable definition within `helper`, not as an assignment within `sort_priority2`:

[Click here to view code image](#)

```
def sort_priority2(numbers, group):
    found = False          # Scope: 'sort_priority2'
    def helper(x):
        if x in group:
            found = True # Scope: 'helper' -- Bad!
            return (0, x)
        return (1, x)

    numbers.sort(key=helper)
    return found
```

This problem is sometimes called the *scoping bug* because it can be so surprising to newbies. But this behavior is the intended result: It prevents local variables in a function from polluting the containing module. Otherwise, every assignment within a function would put garbage into the global module scope. Not only would that be noise, but the interplay of the resulting global variables could cause obscure bugs.

In Python, there is special syntax for getting data out of a closure. The `nonlocal` statement is used to indicate that scope traversal should happen upon assignment for a specific variable name. The only limit is that

`nonlocal` won't traverse up to the module-level scope (to avoid polluting globals).

Here, I define the same function again, now using `nonlocal`:

```
def sort_priority3(numbers, group):
    found = False
    def helper(x):
        nonlocal found # Added
        if x in group:
            found = True
            return (0, x)
        return (1, x)
    numbers.sort(key=helper)
    return found
```

The `nonlocal` statement makes it clear when data is being assigned out of a closure and into another scope. It's complementary to the `global` statement, which indicates that a variable's assignment should go directly into the module scope.

However, much as with the anti-pattern of global variables, I'd caution against using `nonlocal` for anything beyond simple functions. The side effects of `nonlocal` can be hard to follow. It's especially hard to understand in long functions where the `nonlocal` statements and assignments to associated variables are far apart.

When your usage of `nonlocal` starts getting complicated, it's better to wrap your state in a helper class. Here, I define a class that achieves the same result as the `nonlocal` approach; it's a little longer but much easier to read (see [Item 38: "Accept Functions Instead of Classes for Simple Interfaces"](#) for details on the `__call__` special method):

```
class Sorter:
    def __init__(self, group):
        self.group = group
        self.found = False

    def __call__(self, x):
        if x in self.group:
            self.found = True
            return (0, x)
```

```
        return (1, x)

sorter = Sorter(group)
numbers.sort(key=sorter)
assert sorter.found is True
```

## Things to Remember

- ✦ Closure functions can refer to variables from any of the scopes in which they were defined.
- ✦ By default, closures can't affect enclosing scopes by assigning variables.
- ✦ Use the `nonlocal` statement to indicate when a closure can modify a variable in its enclosing scopes.
- ✦ Avoid using `nonlocal` statements for anything beyond simple functions.

## Item 22: Reduce Visual Noise with Variable Positional Arguments

Accepting a variable number of positional arguments can make a function call clearer and reduce visual noise. (These positional arguments are often called *varargs* for short, or *star args*, in reference to the conventional name for the parameter `*args`.) For example, say that I want to log some debugging information. With a fixed number of arguments, I would need a function that takes a message and a list of values:

[Click here to view code image](#)

```
def log(message, values):
    if not values:
        print(message)
    else:
        values_str = ', '.join(str(x) for x in values)
        print(f'{message}: {values_str}')

log('My numbers are', [1, 2])
log('Hi there', [])
```

```
>>>
My numbers are: 1, 2
Hi there
```

Having to pass an empty `list` when I have no values to log is cumbersome and noisy. It'd be better to leave out the second argument entirely. I can do this in Python by prefixing the last positional parameter name with `*`. The first parameter for the log message is required, whereas any number of subsequent positional arguments are optional. The function body doesn't need to change; only the callers do:

[Click here to view code image](#)

```
def log(message, *values): # The only difference
    if not values:
        print(message)
    else:
        values_str = ', '.join(str(x) for x in values)
        print(f'{message}: {values_str}')

log('My numbers are', 1, 2)
log('Hi there') # Much better

>>>
My numbers are: 1, 2
Hi there
```

You might notice that this syntax works very similarly to the starred expressions used in unpacking assignment statements (see [Item 13: “Prefer Catch-All Unpacking Over Slicing”](#)).

If I already have a sequence (like a `list`) and want to call a variadic function like `log`, I can do this by using the `*` operator. This instructs Python to pass items from the sequence as positional arguments to the function:

```
favorites = [7, 33, 99]
log('Favorite colors', *favorites)

>>>
Favorite colors: 7, 33, 99
```

There are two problems with accepting a variable number of positional arguments.

The first issue is that these optional positional arguments are always turned into a tuple before they are passed to a function. This means that if the caller of a function uses the `*` operator on a generator, it will be iterated until it's exhausted (see [Item 30: “Consider Generators Instead of Returning Lists”](#) for background). The resulting tuple includes every value from the generator, which could consume a lot of memory and cause the program to crash:

```
def my_generator():
    for i in range(10):
        yield i

def my_func(*args):
    print(args)

it = my_generator()
my_func(*it)

>>>
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Functions that accept `*args` are best for situations where you know the number of inputs in the argument list will be reasonably small. `*args` is ideal for function calls that pass many literals or variable names together. It's primarily for the convenience of the programmer and the readability of the code.

The second issue with `*args` is that you can't add new positional arguments to a function in the future without migrating every caller. If you try to add a positional argument in the front of the argument list, existing callers will subtly break if they aren't updated:

[Click here to view code image](#)

```
def log(sequence, message, *values):
    if not values:
        print(f'{sequence} - {message}')
    else:
        values_str = ', '.join(str(x) for x in values)
        print(f'{sequence} - {message}: {values_str}')

log(1, 'Favorites', 7, 33)      # New with *args OK
log(1, 'Hi there')             # New message only OK
```

```
log('Favorite numbers', 7, 33) # Old usage breaks
```

```
>>>
```

```
1 - Favorites: 7, 33
```

```
1 - Hi there
```

```
Favorite numbers - 7: 33
```

The problem here is that the third call to `log` used 7 as the message parameter because a sequence argument wasn't given. Bugs like this are hard to track down because the code still runs without raising exceptions. To avoid this possibility entirely, you should use keyword-only arguments when you want to extend functions that accept `*args` (see [Item 25: “Enforce Clarity with Keyword-Only and Positional-Only Arguments”](#)). To be even more defensive, you could also consider using type annotations (see [Item 90: “Consider Static Analysis via typing to Obviate Bugs”](#)).

## Things to Remember

- ◆ Functions can accept a variable number of positional arguments by using `*args` in the `def` statement.
- ◆ You can use the items from a sequence as the positional arguments for a function with the `*` operator.
- ◆ Using the `*` operator with a generator may cause a program to run out of memory and crash.
- ◆ Adding new positional parameters to functions that accept `*args` can introduce hard-to-detect bugs.

## Item 23: Provide Optional Behavior with Keyword Arguments

As in most other programming languages, in Python you may pass arguments by position when calling a function:

```
def remainder(number, divisor):  
    return number % divisor
```

```
assert remainder(20, 7) == 6
```

All normal arguments to Python functions can also be passed by keyword, where the name of the argument is used in an assignment within the parentheses of a function call. The keyword arguments can be passed in any order as long as all of the required positional arguments are specified. You can mix and match keyword and positional arguments. These calls are equivalent:

```
remainder(20, 7)
remainder(20, divisor=7)
remainder(number=20, divisor=7)
remainder(divisor=7, number=20)
```

Positional arguments must be specified before keyword arguments:

[Click here to view code image](#)

```
remainder(number=20, 7)

>>>
Traceback ...
SyntaxError: positional argument follows keyword argument
```

Each argument can be specified only once:

[Click here to view code image](#)

```
remainder(20, number=7)

>>>
Traceback ...
TypeError: remainder() got multiple values for argument
➔ 'number'
```

If you already have a dictionary, and you want to use its contents to call a function like `remainder`, you can do this by using the `**` operator. This instructs Python to pass the values from the dictionary as the corresponding keyword arguments of the function:

```
my_kwargs = {
    'number': 20,
    'divisor': 7,
}
assert remainder(**my_kwargs) == 6
```



You can mix the `**` operator with positional arguments or keyword arguments in the function call, as long as no argument is repeated:

[Click here to view code image](#)

```
my_kwargs = {
    'divisor': 7,
}
assert remainder(number=20, **my_kwargs) == 6
```

You can also use the `**` operator multiple times if you know that the dictionaries don't contain overlapping keys:

[Click here to view code image](#)

```
my_kwargs = {
    'number': 20,
}
other_kwargs = {
    'divisor': 7,
}
assert remainder(**my_kwargs, **other_kwargs) == 6
```

And if you'd like for a function to receive any named keyword argument, you can use the `**kwargs` catch-all parameter to collect those arguments into a dict that you can then process (see [Item 26: “Define Function Decorators with `functools.wraps`”](#) for when this is especially useful):

[Click here to view code image](#)

```
def print_parameters(**kwargs):
    for key, value in kwargs.items():
        print(f'{key} = {value}')

print_parameters(alpha=1.5, beta=9, gamma=4)

>>>
alpha = 1.5
beta = 9
gamma = 4
```

The flexibility of keyword arguments provides three significant benefits.

The first benefit is that keyword arguments make the function call clearer to new readers of the code. With the call `remainder(20, 7)`, it's not evident

which argument is number and which is divisor unless you look at the implementation of the remainder method. In the call with keyword arguments, `number=20` and `divisor=7` make it immediately obvious which parameter is being used for each purpose.

The second benefit of keyword arguments is that they can have default values specified in the function definition. This allows a function to provide additional capabilities when you need them, but you can accept the default behavior most of the time. This eliminates repetitive code and reduces noise.

For example, say that I want to compute the rate of fluid flowing into a vat. If the vat is also on a scale, then I could use the difference between two weight measurements at two different times to determine the flow rate:

[Click here to view code image](#)

```
def flow_rate(weight_diff, time_diff):  
    return weight_diff / time_diff  
  
weight_diff = 0.5  
time_diff = 3  
flow = flow_rate(weight_diff, time_diff)  
print(f'{flow:.3} kg per second')  
  
>>>  
0.167 kg per second
```

In the typical case, it's useful to know the flow rate in kilograms per second. Other times, it'd be helpful to use the last sensor measurements to approximate larger time scales, like hours or days. I can provide this behavior in the same function by adding an argument for the time period scaling factor:

[Click here to view code image](#)

```
def flow_rate(weight_diff, time_diff, period):  
    return (weight_diff / time_diff) * period
```

The problem is that now I need to specify the `period` argument every time I call the function, even in the common case of flow rate per second (where the period is 1):

[Click here to view code image](#)

```
flow_per_second = flow_rate(weight_diff, time_diff, 1)
```

To make this less noisy, I can give the period argument a default value:

[Click here to view code image](#)

```
def flow_rate(weight_diff, time_diff, period=1):  
    return (weight_diff / time_diff) * period
```

The period argument is now optional:

[Click here to view code image](#)

```
flow_per_second = flow_rate(weight_diff, time_diff)  
flow_per_hour = flow_rate(weight_diff, time_diff, period=3600)
```

This works well for simple default values; it gets tricky for complex default values (see [Item 24: “Use None and Docstrings to Specify Dynamic Default Arguments”](#) for details).

The third reason to use keyword arguments is that they provide a powerful way to extend a function’s parameters while remaining backward compatible with existing callers. This means you can provide additional functionality without having to migrate a lot of existing code, which reduces the chance of introducing bugs.

For example, say that I want to extend the `flow_rate` function above to calculate flow rates in weight units besides kilograms. I can do this by adding a new optional parameter that provides a conversion rate to alternative measurement units:

[Click here to view code image](#)

```
def flow_rate(weight_diff, time_diff,  
              period=1, units_per_kg=1):  
    return ((weight_diff * units_per_kg) / time_diff) * period
```

The default argument value for `units_per_kg` is 1, which makes the returned weight units remain kilograms. This means that all existing callers will see no change in behavior. New callers to `flow_rate` can specify the new keyword argument to see the new behavior:

[Click here to view code image](#)

```
pounds_per_hour = flow_rate(weight_diff, time_diff,  
                             period=3600, units_per_kg=2.2)
```

Providing backward compatibility using optional keyword arguments like this is also crucial for functions that accept `*args` (see [Item 22: “Reduce Visual Noise with Variable Positional Arguments”](#)).

The only problem with this approach is that optional keyword arguments like `period` and `units_per_kg` may still be specified as positional arguments:

[Click here to view code image](#)

```
pounds_per_hour = flow_rate(weight_diff, time_diff, 3600, 2.2)
```

Supplying optional arguments positionally can be confusing because it isn’t clear what the values `3600` and `2.2` correspond to. The best practice is to always specify optional arguments using the keyword names and never pass them as positional arguments. As a function author, you can also require that all callers use this more explicit keyword style to minimize potential errors (see [Item 25: “Enforce Clarity with Keyword-Only and Positional-Only Arguments”](#)).

## Things to Remember

- ✦ Function arguments can be specified by position or by keyword.
- ✦ Keywords make it clear what the purpose of each argument is when it would be confusing with only positional arguments.
- ✦ Keyword arguments with default values make it easy to add new behaviors to a function without needing to migrate all existing callers.
- ✦ Optional keyword arguments should always be passed by keyword instead of by position.

## Item 24: Use `None` and Docstrings to Specify Dynamic Default Arguments

Sometimes you need to use a non-static type as a keyword argument's default value. For example, say I want to print logging messages that are marked with the time of the logged event. In the default case, I want the message to include the time when the function was called. I might try the following approach, assuming that the default arguments are reevaluated each time the function is called:

[Click here to view code image](#)

```
from time import sleep
from datetime import datetime

def log(message, when=datetime.now()):
    print(f'{when}: {message}')

log('Hi there!')
sleep(0.1)
log('Hello again!')
```

>>>  
2019-07-06 14:06:15.120124: Hi there!  
2019-07-06 14:06:15.120124: Hello again!

This doesn't work as expected. The timestamps are the same because `datetime.now` is executed only a single time: when the function is defined. A default argument value is evaluated only once per module load, which usually happens when a program starts up. After the module containing this code is loaded, the `datetime.now()` default argument will never be evaluated again.

The convention for achieving the desired result in Python is to provide a default value of `None` and to document the actual behavior in the docstring (see [Item 84: “Write Docstrings for Every Function, Class, and Module”](#) for background). When your code sees the argument value `None`, you allocate the default value accordingly:

[Click here to view code image](#)

```
def log(message, when=None):
    """Log a message with a timestamp.

    Args:
        message: Message to print.
```

```

        when: datetime of when the message occurred.
            Defaults to the present time.
"""
if when is None:
    when = datetime.now()
print(f'{when}: {message}')

```

Now the timestamps will be different:

[Click here to view code image](#)

```

log('Hi there!')
sleep(0.1)
log('Hello again!')

>>>
2019-07-06 14:06:15.222419: Hi there!
2019-07-06 14:06:15.322555: Hello again!

```

Using `None` for default argument values is especially important when the arguments are mutable. For example, say that I want to load a value encoded as JSON data; if decoding the data fails, I want an empty dictionary to be returned by default:

```

import json

def decode(data, default={}):
    try:
        return json.loads(data)
    except ValueError:
        return default

```

The problem here is the same as in the `datetime.now` example above. The dictionary specified for `default` will be shared by all calls to `decode` because default argument values are evaluated only once (at module load time). This can cause extremely surprising behavior:

```

foo = decode('bad data')
foo['stuff'] = 5
bar = decode('also bad')
bar['meep'] = 1
print('Foo:', foo)
print('Bar:', bar)

>>>

```

```
Foo: {'stuff': 5, 'meep': 1}
Bar: {'stuff': 5, 'meep': 1}
```

You might expect two different dictionaries, each with a single key and value. But modifying one seems to also modify the other. The culprit is that `foo` and `bar` are both equal to the `default` parameter. They are the same dictionary object:

```
assert foo is bar
```

The fix is to set the keyword argument default value to `None` and then document the behavior in the function's docstring:

[Click here to view code image](#)

```
def decode(data, default=None):
    """Load JSON data from a string.

    Args:
        data: JSON data to decode.
        default: Value to return if decoding fails.
                Defaults to an empty dictionary.
    """
    try:
        return json.loads(data)
    except ValueError:
        if default is None:
            default = {}
    return default
```

Now, running the same test code as before produces the expected result:

```
foo = decode('bad data')
foo['stuff'] = 5
bar = decode('also bad')
bar['meep'] = 1
print('Foo:', foo)
print('Bar:', bar)
assert foo is not bar
```

```
>>>
Foo: {'stuff': 5}
Bar: {'meep': 1}
```

This approach also works with type annotations (see [Item 90: “Consider Static Analysis via typing to Obviate Bugs”](#)). Here, the `when` argument is

marked as having an optional value that is a datetime. Thus, the only two valid choices for when are None or a datetime object:

[Click here to view code image](#)

```
from typing import Optional

def log_typed(message: str,
              when: Optional[datetime]=None) -> None:
    """Log a message with a timestamp.

    Args:
        message: Message to print.
        when: datetime of when the message occurred.
              Defaults to the present time.
    """
    if when is None:
        when = datetime.now()
    print(f'{when}: {message}')
```

## Things to Remember

- ✦ A default argument value is evaluated only once: during function definition at module load time. This can cause odd behaviors for dynamic values (like {}, [], or datetime.now()).
- ✦ Use None as the default value for any keyword argument that has a dynamic value. Document the actual default behavior in the function's docstring.
- ✦ Using None to represent keyword argument default values also works correctly with type annotations.

## Item 25: Enforce Clarity with Keyword-Only and Positional-Only Arguments

Passing arguments by keyword is a powerful feature of Python functions (see [Item 23: “Provide Optional Behavior with Keyword Arguments”](#)). The flexibility of keyword arguments enables you to write functions that will be clear to new readers of your code for many use cases.



For example, say I want to divide one number by another but know that I need to be very careful about special cases. Sometimes, I want to ignore `ZeroDivisionError` exceptions and return infinity instead. Other times, I want to ignore `OverflowError` exceptions and return zero instead:

[Click here to view code image](#)

```
def safe_division(number, divisor,
                  ignore_overflow,
                  ignore_zero_division):
    try:
        return number / divisor
    except OverflowError:
        if ignore_overflow:
            return 0
        else:
            raise
    except ZeroDivisionError:
        if ignore_zero_division:
            return float('inf')
        else:
            raise
```

Using this function is straightforward. This call ignores the `float` overflow from division and returns zero:

[Click here to view code image](#)

```
result = safe_division(1.0, 10**500, True, False)
print(result)

>>>
0
```

This call ignores the error from dividing by zero and returns infinity:

[Click here to view code image](#)

```
result = safe_division(1.0, 0, False, True)
print(result)

>>>
inf
```

The problem is that it's easy to confuse the position of the two Boolean arguments that control the exception-ignoring behavior. This can easily

cause bugs that are hard to track down. One way to improve the readability of this code is to use keyword arguments. By default, the function can be overly cautious and can always re-raise exceptions:

[Click here to view code image](#)

```
def safe_division_b(number, divisor,
                    ignore_overflow=False,      # Changed
                    ignore_zero_division=False): # Changed
    ...
```

Then, callers can use keyword arguments to specify which of the ignore flags they want to set for specific operations, overriding the default behavior:

[Click here to view code image](#)

```
result = safe_division_b(1.0, 10**500, ignore_overflow=True)
print(result)

result = safe_division_b(1.0, 0, ignore_zero_division=True)
print(result)

>>>
0
inf
```

The problem is, since these keyword arguments are optional behavior, there's nothing forcing callers to use keyword arguments for clarity. Even with the new definition of `safe_division_b`, you can still call it the old way with positional arguments:

[Click here to view code image](#)

```
assert safe_division_b(1.0, 10**500, True, False) == 0
```

With complex functions like this, it's better to require that callers are clear about their intentions by defining functions with *keyword-only arguments*. These arguments can only be supplied by keyword, never by position.

Here, I redefine the `safe_division` function to accept keyword-only arguments. The `*` symbol in the argument list indicates the end of positional arguments and the beginning of keyword-only arguments:

[Click here to view code image](#)

```
def safe_division_c(number, divisor, *, # Changed
                    ignore_overflow=False,
                    ignore_zero_division=False):
    ...
```

Now, calling the function with positional arguments for the keyword arguments won't work:

[Click here to view code image](#)

```
safe_division_c(1.0, 10**500, True, False)
```

```
>>>
```

```
Traceback ...
```

```
TypeError: safe_division_c() takes 2 positional arguments but 4
➡were given
```

But keyword arguments and their default values will work as expected (ignoring an exception in one case and raising it in another):

[Click here to view code image](#)

```
result = safe_division_c(1.0, 0, ignore_zero_division=True)
assert result == float('inf')
```

```
try:
```

```
    result = safe_division_c(1.0, 0)
```

```
except ZeroDivisionError:
```

```
    pass # Expected
```

However, a problem still remains with the `safe_division_c` version of this function: Callers may specify the first two required arguments (number and divisor) with a mix of positions and keywords:

[Click here to view code image](#)

```
assert safe_division_c(number=2, divisor=5) == 0.4
assert safe_division_c(divisor=5, number=2) == 0.4
assert safe_division_c(2, divisor=5) == 0.4
```

Later, I may decide to change the names of these first two arguments because of expanding needs or even just because my style preferences change:

[Click here to view code image](#)

```
def safe_division_c(numerator, denominator, *, # Changed
                    ignore_overflow=False,
                    ignore_zero_division=False):
    ...
```

Unfortunately, this seemingly superficial change breaks all the existing callers that specified the number or divisor arguments using keywords:

[Click here to view code image](#)

```
safe_division_c(number=2, divisor=5)

>>>
Traceback ...
TypeError: safe_division_c() got an unexpected keyword argument
➔ 'number'
```

This is especially problematic because I never intended for `number` and `divisor` to be part of an explicit interface for this function. These were just convenient parameter names that I chose for the implementation, and I didn't expect anyone to rely on them explicitly.

Python 3.8 introduces a solution to this problem, called *positional-only arguments*. These arguments can be supplied only by position and never by keyword (the opposite of the keyword-only arguments demonstrated above).

Here, I redefine the `safe_division` function to use positional-only arguments for the first two required parameters. The `/` symbol in the argument list indicates where positional-only arguments end:

[Click here to view code image](#)

```
def safe_division_d(numerator, denominator, /, *, # Changed
                    ignore_overflow=False,
                    ignore_zero_division=False):
    ...
```

I can verify that this function works when the required arguments are provided positionally:

```
assert safe_division_d(2, 5) == 0.4
```

But an exception is raised if keywords are used for the positional-only parameters:

[Click here to view code image](#)

```
safe_division_d(numerator=2, denominator=5)

>>>
Traceback ...
TypeError: safe_division_d() got some positional-only arguments
➡passed as keyword arguments: 'numerator, denominator'
```

Now, I can be sure that the first two required positional arguments in the definition of the `safe_division_d` function are decoupled from callers. I won't break anyone if I change the parameters' names again.

One notable consequence of keyword- and positional-only arguments is that any parameter name between the `/` and `*` symbols in the argument list may be passed either by position or by keyword (which is the default for all function arguments in Python). Depending on your API's style and needs, allowing both argument passing styles can increase readability and reduce noise. For example, here I've added another optional parameter to `safe_division` that allows callers to specify how many digits to use in rounding the result:

[Click here to view code image](#)

```
def safe_division_e(numerator, denominator, /,
                    ndigits=10, *,
                    ignore_overflow=False,
                    ignore_zero_division=False):
    try:
        fraction = numerator / denominator
        return round(fraction, ndigits)
    except OverflowError:
        if ignore_overflow:
            return 0
        else:
            raise
    except ZeroDivisionError:
        if ignore_zero_division:
            return float('inf')
        else:
            raise
```

Now, I can call this new version of the function in all these different ways, since `ndigits` is an optional parameter that may be passed either by position or by keyword:

[Click here to view code image](#)

```
result = safe_division_e(22, 7)
print(result)

result = safe_division_e(22, 7, 5)
print(result)

result = safe_division_e(22, 7, ndigits=2)
print(result)

>>>
3.1428571429
3.14286
3.14
```

## Things to Remember

- ◆ Keyword-only arguments force callers to supply certain arguments by keyword (instead of by position), which makes the intention of a function call clearer. Keyword-only arguments are defined after a single `*` in the argument list.
- ◆ Positional-only arguments ensure that callers can't supply certain parameters using keywords, which helps reduce coupling. Positional-only arguments are defined before a single `/` in the argument list.
- ◆ Parameters between the `/` and `*` characters in the argument list may be supplied by position or keyword, which is the default for Python parameters.

## Item 26: Define Function Decorators with `functools.wraps`

Python has special syntax for *decorators* that can be applied to functions. A decorator has the ability to run additional code before and after each call to a function it wraps. This means decorators can access and modify input

arguments, return values, and raised exceptions. This functionality can be useful for enforcing semantics, debugging, registering functions, and more.

For example, say that I want to print the arguments and return value of a function call. This can be especially helpful when debugging the stack of nested function calls from a recursive function. Here, I define such a decorator by using `*args` and `**kwargs` (see [Item 22: “Reduce Visual Noise with Variable Positional Arguments”](#) and [Item 23: “Provide Optional Behavior with Keyword Arguments”](#)) to pass through all parameters to the wrapped function:

[Click here to view code image](#)

```
def trace(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        print(f'{func.__name__}({args!r}, {kwargs!r}) '
              f'-> {result!r}')
        return result
    return wrapper
```

I can apply this decorator to a function by using the `@` symbol:

[Click here to view code image](#)

```
@trace
def fibonacci(n):
    """Return the n-th Fibonacci number"""
    if n in (0, 1):
        return n
    return (fibonacci(n - 2) + fibonacci(n - 1))
```

Using the `@` symbol is equivalent to calling the decorator on the function it wraps and assigning the return value to the original name in the same scope:

```
fibonacci = trace(fibonacci)
```

The decorated function runs the wrapper code before and after `fibonacci` runs. It prints the arguments and return value at each level in the recursive stack:

```
fibonacci(4)
```

```
>>>
```

```
fibonacci((0,), {}) -> 0
fibonacci((1,), {}) -> 1
fibonacci((2,), {}) -> 1
fibonacci((1,), {}) -> 1
fibonacci((0,), {}) -> 0
fibonacci((1,), {}) -> 1
fibonacci((2,), {}) -> 1
fibonacci((3,), {}) -> 2
fibonacci((4,), {}) -> 3
```

This works well, but it has an unintended side effect. The value returned by the decorator—the function that’s called above—doesn’t think it’s named `fibonacci`:

[Click here to view code image](#)

```
print(fibonacci)

>>>
<function trace.<locals>.wrapper at 0x108955dc0>
```

The cause of this isn’t hard to see. The `trace` function returns the `wrapper` defined within its body. The `wrapper` function is what’s assigned to the `fibonacci` name in the containing module because of the decorator. This behavior is problematic because it undermines tools that do introspection, such as debuggers (see [Item 80: “Consider Interactive Debugging with `pdb`”](#)).

For example, the `help` built-in function is useless when called on the decorated `fibonacci` function. It should instead print out the docstring defined above (`'Return the n-th Fibonacci number'`):

[Click here to view code image](#)

```
help(fibonacci)

>>>
Help on function wrapper in module __main__:

wrapper(*args, **kwargs)
```

Object serializers (see [Item 68: “Make `pickle` Reliable with `copyreg`”](#)) break because they can’t determine the location of the original function that



was decorated:

[Click here to view code image](#)

```
import pickle

pickle.dumps(fibonacci)

>>>
Traceback ...
AttributeError: Can't pickle local object 'trace.<locals>.
➔wrapper'
```

The solution is to use the `wraps` helper function from the `functools` built-in module. This is a decorator that helps you write decorators. When you apply it to the `wrapper` function, it copies all of the important metadata about the inner function to the outer function:

```
from functools import wraps

def trace(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        ...
    return wrapper
@trace
def fibonacci(n):
    ...
```

Now, running the `help` function produces the expected result, even though the function is decorated:

[Click here to view code image](#)

```
help(fibonacci)

>>>
Help on function fibonacci in module __main__:
fibonacci(n)
    Return the n-th Fibonacci number
```

The `pickle` object serializer also works:

[Click here to view code image](#)

```
print(pickle.dumps(fibonacci))
```

```
>>>
```

```
b'\x80\x04\x95\x1a\x00\x00\x00\x00\x00\x00\x00\x8c\x08__main__\n\x94\x8c\tfibonacci\x94\x93\x94.'
```

Beyond these examples, Python functions have many other standard attributes (e.g., `__name__`, `__module__`, `__annotations__`) that must be preserved to maintain the interface of functions in the language. Using `wraps` ensures that you'll always get the correct behavior.

## Things to Remember

- ◆ Decorators in Python are syntax to allow one function to modify another function at runtime.
- ◆ Using decorators can cause strange behaviors in tools that do introspection, such as debuggers.
- ◆ Use the `wraps` decorator from the `functools` built-in module when you define your own decorators to avoid issues.

## 4. Comprehensions and Generators

Many programs are built around processing lists, dictionary key/value pairs, and sets. Python provides a special syntax, called *comprehensions*, for succinctly iterating through these types and creating derivative data structures. Comprehensions can significantly increase the readability of code performing these common tasks and provide a number of other benefits.

This style of processing is extended to functions with *generators*, which enable a stream of values to be incrementally returned by a function. The result of a call to a generator function can be used anywhere an iterator is appropriate (e.g., for loops, starred expressions). Generators can improve performance, reduce memory usage, and increase readability.

### Item 27: Use Comprehensions Instead of `map` and `filter`

Python provides compact syntax for deriving a new list from another sequence or iterable. These expressions are called *list comprehensions*. For example, say that I want to compute the square of each number in a list. Here, I do this by using a simple for loop:

[Click here to view code image](#)

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
squares = []
for x in a:
    squares.append(x**2)
print(squares)

>>>
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

With a list comprehension, I can achieve the same outcome by specifying the expression for my computation along with the input sequence to loop over:

[Click here to view code image](#)

```
squares = [x**2 for x in a] # List comprehension
print(squares)
```

```
>>>
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Unless you're applying a single-argument function, list comprehensions are also clearer than the `map` built-in function for simple cases. `map` requires the creation of a `lambda` function for the computation, which is visually noisy:

```
alt = map(lambda x: x ** 2, a)
```

Unlike `map`, list comprehensions let you easily filter items from the input list, removing corresponding outputs from the result. For example, say I want to compute the squares of the numbers that are divisible by 2. Here, I do this by adding a conditional expression to the list comprehension after the loop:

[Click here to view code image](#)

```
even_squares = [x**2 for x in a if x % 2 == 0]
print(even_squares)
```

```
>>>
[4, 16, 36, 64, 100]
```

The `filter` built-in function can be used along with `map` to achieve the same outcome, but it is much harder to read:

[Click here to view code image](#)

```
alt = map(lambda x: x**2, filter(lambda x: x % 2 == 0, a))
assert even_squares == list(alt)
```

Dictionaries and sets have their own equivalents of list comprehensions (called *dictionary comprehensions* and *set comprehensions*, respectively). These make it easy to create other types of derivative data structures when writing algorithms:

[Click here to view code image](#)

```
even_squares_dict = {x: x**2 for x in a if x % 2 == 0}
threes_cubed_set = {x**3 for x in a if x % 3 == 0}
print(even_squares_dict)
```

```
print(threes_cubed_set)

>>>
{2: 4, 4: 16, 6: 36, 8: 64, 10: 100}
{216, 729, 27}
```

Achieving the same outcome is possible with `map` and `filter` if you wrap each call with a corresponding constructor. These statements get so long that you have to break them up across multiple lines, which is even noisier and should be avoided:

[Click here to view code image](#)

```
alt_dict = dict(map(lambda x: (x, x**2),
                    filter(lambda x: x % 2 == 0, a)))
alt_set = set(map(lambda x: x**3,
                  filter(lambda x: x % 3 == 0, a)))
```

## Things to Remember

- ◆ List comprehensions are clearer than the `map` and `filter` built-in functions because they don't require `lambda` expressions.
- ◆ List comprehensions allow you to easily skip items from the input `list`, a behavior that `map` doesn't support without help from `filter`.
- ◆ Dictionaries and sets may also be created using comprehensions.

## Item 28: Avoid More Than Two Control Subexpressions in Comprehensions

Beyond basic usage (see [Item 27: “Use Comprehensions Instead of `map` and `filter`”](#)), comprehensions support multiple levels of looping. For example, say that I want to simplify a matrix (a `list` containing other `list` instances) into one flat `list` of all cells. Here, I do this with a list comprehension by including two `for` subexpressions. These subexpressions run in the order provided, from left to right:

[Click here to view code image](#)

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flat = [x for row in matrix for x in row]
print(flat)
```

```
>>>
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

This example is simple, readable, and a reasonable usage of multiple loops in a comprehension. Another reasonable usage of multiple loops involves replicating the two-level-deep layout of the input `list`. For example, say that I want to square the value in each cell of a twodimensional matrix. This comprehension is noisier because of the extra `[]` characters, but it's still relatively easy to read:

[Click here to view code image](#)

```
squared = [[x**2 for x in row] for row in matrix]
print(squared)
```

```
>>>
[[1, 4, 9], [16, 25, 36], [49, 64, 81]]
```

If this comprehension included another loop, it would get so long that I'd have to split it over multiple lines:

```
my_lists = [
    [[1, 2, 3], [4, 5, 6]],
    ...
]
flat = [x for sublist1 in my_lists
        for sublist2 in sublist1
        for x in sublist2]
```

At this point, the multiline comprehension isn't much shorter than the alternative. Here, I produce the same result using normal loop statements. The indentation of this version makes the looping clearer than the three-level-list comprehension:

```
flat = []
for sublist1 in my_lists:
    for sublist2 in sublist1:
        flat.extend(sublist2)
```

Comprehensions support multiple `if` conditions. Multiple conditions at the same loop level have an implicit `and` expression. For example, say that I want to filter a list of numbers to only even values greater than 4. These two list comprehensions are equivalent:

[Click here to view code image](#)

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
b = [x for x in a if x > 4 if x % 2 == 0]
c = [x for x in a if x > 4 and x % 2 == 0]
```

Conditions can be specified at each level of looping after the `for` subexpression. For example, say I want to filter a matrix so the only cells remaining are those divisible by 3 in rows that sum to 10 or higher. Expressing this with a list comprehension does not require a lot of code, but it is extremely difficult to read:

[Click here to view code image](#)

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
filtered = [[x for x in row if x % 3 == 0]
            for row in matrix if sum(row) >= 10]
print(filtered)

>>>
[[6], [9]]
```

Although this example is a bit convoluted, in practice you'll see situations arise where such comprehensions seem like a good fit. I strongly encourage you to avoid using list, dict, or set comprehensions that look like this. The resulting code is very difficult for new readers to understand. The potential for confusion is even worse for dict comprehensions since they already need an extra parameter to represent both the key and the value for each item.

The rule of thumb is to avoid using more than two control subexpressions in a comprehension. This could be two conditions, two loops, or one condition and one loop. As soon as it gets more complicated than that, you should use normal `if` and `for` statements and write a helper function (see [Item 30](#): “Consider Generators Instead of Returning Lists”).

## Things to Remember

- ✦ Comprehensions support multiple levels of loops and multiple conditions per loop level.
- ✦ Comprehensions with more than two control subexpressions are very difficult to read and should be avoided.

## Item 29: Avoid Repeated Work in Comprehensions by Using Assignment Expressions

A common pattern with comprehensions—including `list`, `dict`, and `set` variants—is the need to reference the same computation in multiple places. For example, say that I’m writing a program to manage orders for a fastener company. As new orders come in from customers, I need to be able to tell them whether I can fulfill their orders. I need to verify that a request is sufficiently in stock and above the minimum threshold for shipping (in batches of 8):

[Click here to view code image](#)

```
stock = {
    'nails': 125,
    'screws': 35,
    'wingnuts': 8,
    'washers': 24,
}

order = ['screws', 'wingnuts', 'clips']

def get_batches(count, size):
    return count // size

result = {}
for name in order:
    count = stock.get(name, 0)
    batches = get_batches(count, 8)
    if batches:
        result[name] = batches
print(result)
```



```
>>>
{'screws': 4, 'wingnuts': 1}
```

Here, I implement this looping logic more succinctly using a dictionary comprehension (see [Item 27: “Use Comprehensions Instead of map and filter”](#) for best practices):

[Click here to view code image](#)

```
found = {name: get_batches(stock.get(name, 0), 8)
         for name in order
         if get_batches(stock.get(name, 0), 8)}
print(found)

>>>
{'screws': 4, 'wingnuts': 1}
```

Although this code is more compact, the problem with it is that the `get_batches(stock.get(name, 0), 8)` expression is repeated. This hurts readability by adding visual noise that’s technically unnecessary. It also increases the likelihood of introducing a bug if the two expressions aren’t kept in sync. For example, here I’ve changed the first `get_batches` call to have 4 as its second parameter instead of 8, which causes the results to be different:

[Click here to view code image](#)

```
has_bug = {name: get_batches(stock.get(name, 0), 4)
          for name in order
          if get_batches(stock.get(name, 0), 8)}

print('Expected:', found)
print('Found: ', has_bug)

>>>
Expected: {'screws': 4, 'wingnuts': 1}
Found: {'screws': 8, 'wingnuts': 2}
```

An easy solution to these problems is to use the walrus operator (`:=`), which was introduced in Python 3.8, to form an assignment expression as part of the comprehension (see [Item 10: “Prevent Repetition with Assignment Expressions”](#) for background):

[Click here to view code image](#)

```
found = {name: batches for name in order
         if (batches := get_batches(stock.get(name, 0), 8))}
```

The assignment expression (`batches := get_batches(...)`) allows me to look up the value for each order key in the stock dictionary a single time, call `get_batches` once, and then store its corresponding value in the `batches` variable. I can then reference that variable elsewhere in the comprehension to construct the dict's contents instead of having to call `get_batches` a second time. Eliminating the redundant calls to `get` and `get_batches` may also improve performance by avoiding unnecessary computations for each item in the `order` list.

It's valid syntax to define an assignment expression in the value expression for a comprehension. But if you try to reference the variable it defines in other parts of the comprehension, you might get an exception at runtime because of the order in which comprehensions are evaluated:

[Click here to view code image](#)

```
result = {name: (tenth := count // 10)
          for name, count in stock.items() if tenth > 0}
```

```
>>>
Traceback ...
NameError: name 'tenth' is not defined
```

I can fix this example by moving the assignment expression into the condition and then referencing the variable name it defined in the comprehension's value expression:

[Click here to view code image](#)

```
result = {name: tenth for name, count in stock.items()
          if (tenth := count // 10) > 0}
print(result)
```

```
>>>
{'nails': 12, 'screws': 3, 'washers': 2}
```

If a comprehension uses the walrus operator in the value part of the comprehension and doesn't have a condition, it'll leak the loop variable into

the containing scope (see [Item 21: “Know How Closures Interact with Variable Scope”](#) for background):

[Click here to view code image](#)

```
half = [(last := count // 2) for count in stock.values()]
print(f'Last item of {half} is {last}')

>>>
Last item of [62, 17, 4, 12] is 12
```

This leakage of the loop variable is similar to what happens with a normal for loop:

[Click here to view code image](#)

```
for count in stock.values(): # Leaks loop variable
    pass
print(f'Last item of {list(stock.values())} is {count}')
```

```
>>>
Last item of [125, 35, 8, 24] is 24
```

However, similar leakage doesn't happen for the loop variables from comprehensions:

[Click here to view code image](#)

```
half = [count // 2 for count in stock.values()]
print(half) # Works
print(count) # Exception because loop variable didn't leak

>>>
[62, 17, 4, 12]
Traceback ...
NameError: name 'count' is not defined
```

It's better not to leak loop variables, so I recommend using assignment expressions only in the condition part of a comprehension.

Using an assignment expression also works the same way in generator expressions (see [Item 32: “Consider Generator Expressions for Large List Comprehensions”](#)). Here, I create an iterator of pairs containing the item name and the current count in stock instead of a dict instance:

[Click here to view code image](#)

```
found = ((name, batches) for name in order
         if (batches := get_batches(stock.get(name, 0), 8)))
print(next(found))
print(next(found))

>>>
('screws', 4)
('wingnuts', 1)
```

## Things to Remember

- ✦ Assignment expressions make it possible for comprehensions and generator expressions to reuse the value from one condition elsewhere in the same comprehension, which can improve readability and performance.
- ✦ Although it's possible to use an assignment expression outside of a comprehension or generator expression's condition, you should avoid doing so.

## Item 30: Consider Generators Instead of Returning Lists

The simplest choice for a function that produces a sequence of results is to return a list of items. For example, say that I want to find the index of every word in a string. Here, I accumulate results in a list using the `append` method and return it at the end of the function:

[Click here to view code image](#)

```
def index_words(text):
    result = []
    if text:
        result.append(0)
    for index, letter in enumerate(text):
        if letter == ' ':
            result.append(index + 1)
    return result
```

This works as expected for some sample input:

[Click here to view code image](#)

```

address = 'Four score and seven years ago...'
result = index_words(address)
print(result[:10])

>>>
[0, 5, 11, 15, 21, 27, 31, 35, 43, 51]

```

There are two problems with the `index_words` function.

The first problem is that the code is a bit dense and noisy. Each time a new result is found, I call the `append` method. The method call's bulk (`result.append`) deemphasizes the value being added to the list (`index + 1`). There is one line for creating the result list and another for returning it. While the function body contains ~130 characters (without whitespace), only ~75 characters are important.

A better way to write this function is by using a *generator*. Generators are produced by functions that use `yield` expressions. Here, I define a generator function that produces the same results as before:

[Click here to view code image](#)

```

def index_words_iter(text):
    if text:
        yield 0
    for index, letter in enumerate(text):
        if letter == ' ':
            yield index + 1

```

When called, a generator function does not actually run but instead immediately returns an iterator. With each call to the `next` built-in function, the iterator advances the generator to its next `yield` expression. Each value passed to `yield` by the generator is returned by the iterator to the caller:

```

it = index_words_iter(address)
print(next(it))
print(next(it))
>>>
0
5

```

The `index_words_iter` function is significantly easier to read because all interactions with the result list have been eliminated. Results are passed to

yield expressions instead. You can easily convert the iterator returned by the generator to a list by passing it to the `list` built-in function if necessary (see [Item 32: “Consider Generator Expressions for Large List Comprehensions”](#) for how this works):

[Click here to view code image](#)

```
result = list(index_words_iter(address))
print(result[:10])
```

```
>>>
[0, 5, 11, 15, 21, 27, 31, 35, 43, 51]
```

The second problem with `index_words` is that it requires all results to be stored in the list before being returned. For huge inputs, this can cause a program to run out of memory and crash.

In contrast, a generator version of this function can easily be adapted to take inputs of arbitrary length due to its bounded memory requirements. For example, here I define a generator that streams input from a file one line at a time and yields outputs one word at a time:

```
def index_file(handle):
    offset = 0
    for line in handle:
        if line:
            yield offset
            for letter in line:
                offset += 1
                if letter == ' ':
                    yield offset
```

The working memory for this function is limited to the maximum length of one line of input. Running the generator produces the same results (see [Item 36: “Consider `itertools` for Working with Iterators and Generators”](#) for more about the `islice` function):

[Click here to view code image](#)

```
with open('address.txt', 'r') as f:
    it = index_file(f)
    results = itertools.islice(it, 0, 10)
    print(list(results))
```

```
>>>
[0, 5, 11, 15, 21, 27, 31, 35, 43, 51]
```

The only gotcha with defining generators like this is that the callers must be aware that the iterators returned are stateful and can't be reused (see [Item 31: “Be Defensive When Iterating Over Arguments”](#)).

## Things to Remember

- ✦ Using generators can be clearer than the alternative of having a function return a list of accumulated results.
- ✦ The iterator returned by a generator produces the set of values passed to yield expressions within the generator function's body.
- ✦ Generators can produce a sequence of outputs for arbitrarily large inputs because their working memory doesn't include all inputs and outputs.

## Item 31: Be Defensive When Iterating Over Arguments

When a function takes a list of objects as a parameter, it's often important to iterate over that list multiple times. For example, say that I want to analyze tourism numbers for the U.S. state of Texas. Imagine that the data set is the number of visitors to each city (in millions per year). I'd like to figure out what percentage of overall tourism each city receives.

To do this, I need a normalization function that sums the inputs to determine the total number of tourists per year and then divides each city's individual visitor count by the total to find that city's contribution to the whole:

```
def normalize(numbers):
    total = sum(numbers)
    result = []
    for value in numbers:
        percent = 100 * value / total
        result.append(percent)
    return result
```

This function works as expected when given a list of visits:

[Click here to view code image](#)

```
visits = [15, 35, 80]
percentages = normalize(visits)
print(percentages)
assert sum(percentages) == 100.0
```

```
>>>
[11.538461538461538, 26.923076923076923, 61.53846153846154]
```

To scale this up, I need to read the data from a file that contains every city in all of Texas. I define a generator to do this because then I can reuse the same function later, when I want to compute tourism numbers for the whole world—a much larger data set with higher memory requirements (see [Item 30: “Consider Generators Instead of Returning Lists”](#) for background):

```
def read_visits(data_path):
    with open(data_path) as f:
        for line in f:
            yield int(line)
```

Surprisingly, calling `normalize` on the `read_visits` generator’s return value produces no results:

```
it = read_visits('my_numbers.txt')
percentages = normalize(it)
print(percentages)
```

```
>>>
[]
```

This behavior occurs because an iterator produces its results only a single time. If you iterate over an iterator or a generator that has already raised a `StopIteration` exception, you won’t get any results the second time around:

```
it = read_visits('my_numbers.txt')
print(list(it))
print(list(it)) # Already exhausted
```

```
>>>
[15, 35, 80]
[]
```

Confusingly, you also won’t get errors when you iterate over an already exhausted iterator. For loops, the `list` constructor, and many other functions



throughout the Python standard library expect the `StopIteration` exception to be raised during normal operation. These functions can't tell the difference between an iterator that has no output and an iterator that had output and is now exhausted.

To solve this problem, you can explicitly exhaust an input iterator and keep a copy of its entire contents in a `list`. You can then iterate over the `list` version of the data as many times as you need to. Here's the same function as before, but it defensively copies the input iterator:

[Click here to view code image](#)

```
def normalize_copy(numbers):
    numbers_copy = list(numbers) # Copy the iterator
    total = sum(numbers_copy)
    result = []
    for value in numbers_copy:
        percent = 100 * value / total
        result.append(percent)
    return result
```

Now the function works correctly on the `read_visits` generator's return value:

[Click here to view code image](#)

```
it = read_visits('my_numbers.txt')
percentages = normalize_copy(it)
print(percentages)
assert sum(percentages) == 100.0
```

```
>>>
[11.538461538461538, 26.923076923076923, 61.53846153846154]
```

The problem with this approach is that the copy of the input iterator's contents could be extremely large. Copying the iterator could cause the program to run out of memory and crash. This potential for scalability issues undermines the reason that I wrote `read_visits` as a generator in the first place. One way around this is to accept a function that returns a new iterator each time it's called:

[Click here to view code image](#)

```
def normalize_func(get_iter):
    total = sum(get_iter()) # New iterator
    result = []
    for value in get_iter(): # New iterator
        percent = 100 * value / total
        result.append(percent)
    return result
```

To use `normalize_func`, I can pass in a lambda expression that calls the generator and produces a new iterator each time:

[Click here to view code image](#)

```
path = 'my_numbers.txt'
percentages = normalize_func(lambda: read_visits(path))
print(percentages)
assert sum(percentages) == 100.0

>>>
[11.538461538461538, 26.923076923076923, 61.53846153846154]
```

Although this works, having to pass a lambda function like this is clumsy. A better way to achieve the same result is to provide a new container class that implements the *iterator protocol*.

The iterator protocol is how Python for loops and related expressions traverse the contents of a container type. When Python sees a statement like `for x in foo`, it actually calls `iter(foo)`. The `iter` built-in function calls the `foo.__iter__` special method in turn. The `__iter__` method must return an iterator object (which itself implements the `__next__` special method). Then, the `for` loop repeatedly calls the `next` built-in function on the iterator object until it's exhausted (indicated by raising a `StopIteration` exception).

It sounds complicated, but practically speaking, you can achieve all of this behavior for your classes by implementing the `__iter__` method as a generator. Here, I define an iterable container class that reads the file containing tourism data:

```
class ReadVisits:
    def __init__(self, data_path):
        self.data_path = data_path

    def __iter__(self):
```

```
with open(self.data_path) as f:
    for line in f:
        yield int(line)
```

This new container type works correctly when passed to the original function without modifications:

[Click here to view code image](#)

```
visits = ReadVisits(path)
percentages = normalize(visits)
print(percentages)
assert sum(percentages) == 100.0
```

```
>>>
[11.538461538461538, 26.923076923076923, 61.53846153846154]
```

This works because the `sum` method in `normalize` calls `ReadVisits.__iter__` to allocate a new iterator object. The `for` loop to normalize the numbers also calls `__iter__` to allocate a second iterator object. Each of those iterators will be advanced and exhausted independently, ensuring that each unique iteration sees all of the input data values. The only downside of this approach is that it reads the input data multiple times.

Now that you know how containers like `ReadVisits` work, you can write your functions and methods to ensure that parameters aren't just iterators. The protocol states that when an iterator is passed to the `iter` built-in function, `iter` returns the iterator itself. In contrast, when a container type is passed to `iter`, a new iterator object is returned each time. Thus, you can test an input value for this behavior and raise a `TypeError` to reject arguments that can't be repeatedly iterated over:

[Click here to view code image](#)

```
def normalize_defensive(numbers):
    if iter(numbers) is numbers: # An iterator -- bad!
        raise TypeError('Must supply a container')
    total = sum(numbers)
    result = []
    for value in numbers:
        percent = 100 * value / total
        result.append(percent)
    return result
```

Alternatively, the `collections.abc` built-in module defines an `Iterator` class that can be used in an `isinstance` test to recognize the potential problem (see [Item 43: “Inherit from `collections.abc` for Custom Container Types”](#)):

[Click here to view code image](#)

```
from collections.abc import Iterator

def normalize_defensive(numbers):
    if isinstance(numbers, Iterator): # Another way to check
        raise TypeError('Must supply a container')
    total = sum(numbers)
    result = []
    for value in numbers:
        percent = 100 * value / total
        result.append(percent)
    return result
```

The approach of using a container is ideal if you don’t want to copy the full input iterator, as with the `normalize_copy` function above, but you also need to iterate over the input data multiple times. This function works as expected for `list` and `ReadVisits` inputs because they are iterable containers that follow the iterator protocol:

[Click here to view code image](#)

```
visits = [15, 35, 80]
percentages = normalize_defensive(visits)
assert sum(percentages) == 100.0

visits = ReadVisits(path)
percentages = normalize_defensive(visits)
assert sum(percentages) == 100.0
```

The function raises an exception if the input is an iterator rather than a container:

```
visits = [15, 35, 80]
it = iter(visits)
normalize_defensive(it)

>>>
Traceback ...
TypeError: Must supply a container
```

The same approach can also be used for asynchronous iterators (see [Item 61: “Know How to Port Threaded I/O to `asyncio`”](#) for an example).

## Things to Remember

- ✦ Beware of functions and methods that iterate over input arguments multiple times. If these arguments are iterators, you may see strange behavior and missing values.
- ✦ Python’s iterator protocol defines how containers and iterators interact with the `iter` and `next` built-in functions, `for` loops, and related expressions.
- ✦ You can easily define your own iterable container type by implementing the `__iter__` method as a generator.
- ✦ You can detect that a value is an iterator (instead of a container) if calling `iter` on it produces the same value as what you passed in. Alternatively, you can use the `isinstance` built-in function along with the `collections.abc.Iterator` class.

## Item 32: Consider Generator Expressions for Large List Comprehensions

The problem with list comprehensions (see [Item 27: “Use Comprehensions Instead of `map` and `filter`”](#)) is that they may create new `list` instances containing one item for each value in input sequences. This is fine for small inputs, but for large inputs, this behavior could consume significant amounts of memory and cause a program to crash.

For example, say that I want to read a file and return the number of characters on each line. Doing this with a list comprehension would require holding the length of every line of the file in memory. If the file is enormous or perhaps a never-ending network socket, using list comprehensions would be problematic. Here, I use a list comprehension in a way that can only handle small input values:

[Click here to view code image](#)

```
value = [len(x) for x in open('my_file.txt')]
print(value)
```

```
>>>
[100, 57, 15, 1, 12, 75, 5, 86, 89, 11]
```

To solve this issue, Python provides *generator expressions*, which are a generalization of list comprehensions and generators. Generator expressions don't materialize the whole output sequence when they're run. Instead, generator expressions evaluate to an iterator that yields one item at a time from the expression.

You create a generator expression by putting list-comprehension-like syntax between `()` characters. Here, I use a generator expression that is equivalent to the code above. However, the generator expression immediately evaluates to an iterator and doesn't make forward progress:

[Click here to view code image](#)

```
it = (len(x) for x in open('my_file.txt'))
print(it)
```

```
>>>
<generator object <genexpr> at 0x108993dd0>
```

The returned iterator can be advanced one step at a time to produce the next output from the generator expression, as needed (using the `next` built-in function). I can consume as much of the generator expression as I want without risking a blowup in memory usage:

```
print(next(it))
print(next(it))
```

```
>>>
100
57
```

Another powerful outcome of generator expressions is that they can be composed together. Here, I take the iterator returned by the generator expression above and use it as the input for another generator expression:

```
roots = ((x, x**0.5) for x in it)
```

Each time I advance this iterator, it also advances the interior iterator, creating a domino effect of looping, evaluating conditional expressions, and passing around inputs and outputs, all while being as memory efficient as possible:

```
print(next(roots))
```

```
>>>
```

```
(15, 3.872983346207417)
```

Chaining generators together like this executes very quickly in Python. When you're looking for a way to compose functionality that's operating on a large stream of input, generator expressions are a great choice. The only gotcha is that the iterators returned by generator expressions are stateful, so you must be careful not to use these iterators more than once (see [Item 31: “Be Defensive When Iterating Over Arguments”](#)).

## Things to Remember

- ✦ List comprehensions can cause problems for large inputs by using too much memory.
- ✦ Generator expressions avoid memory issues by producing outputs one at a time as iterators.
- ✦ Generator expressions can be composed by passing the iterator from one generator expression into the `for` subexpression of another.
- ✦ Generator expressions execute very quickly when chained together and are memory efficient.

## Item 33: Compose Multiple Generators with `yield from`

Generators provide a variety of benefits (see [Item 30: “Consider Generators Instead of Returning Lists”](#)) and solutions to common problems (see [Item 31: “Be Defensive When Iterating Over Arguments”](#)). Generators are so useful that many programs start to look like layers of generators strung together.

For example, say that I have a graphical program that's using generators to animate the movement of images onscreen. To get the visual effect I'm looking for, I need the images to move quickly at first, pause temporarily, and then continue moving at a slower pace. Here, I define two generators that yield the expected onscreen deltas for each part of this animation:

```
def move(period, speed):
    for _ in range(period):
        yield speed

def pause(delay):
    for _ in range(delay):
        yield 0
```

To create the final animation, I need to combine `move` and `pause` together to produce a single sequence of onscreen deltas. Here, I do this by calling a generator for each step of the animation, iterating over each generator in turn, and then yielding the deltas from all of them in sequence:

```
def animate():
    for delta in move(4, 5.0):
        yield delta
    for delta in pause(3):
        yield delta
    for delta in move(2, 3.0):
        yield delta
```

Now, I can render those deltas onscreen as they're produced by the single animation generator:

```
def render(delta):
    print(f'Delta: {delta:.1f}')
    # Move the images onscreen
    ...

def run(func):
    for delta in func():
        render(delta)
```

```
run(animate)
```

```
>>>
```

```
Delta: 5.0
```

```
Delta: 5.0
```

```
Delta: 5.0
```



```
Delta: 5.0
Delta: 0.0
Delta: 0.0
Delta: 0.0
Delta: 3.0
Delta: 3.0
```

The problem with this code is the repetitive nature of the `animate` function. The redundancy of the `for` statements and `yield` expressions for each generator adds noise and reduces readability. This example includes only three nested generators and it's already hurting clarity; a complex animation with a dozen phases or more would be extremely difficult to follow.

The solution to this problem is to use the `yield from` expression. This advanced generator feature allows you to yield all values from a nested generator before returning control to the parent generator. Here, I reimplement the animation function by using `yield from`:

```
def animate_composed():
    yield from move(4, 5.0)
    yield from pause(3)
    yield from move(2, 3.0)
```

```
run(animate_composed)
```

```
>>>
Delta: 5.0
Delta: 5.0
Delta: 5.0
Delta: 5.0
Delta: 0.0
Delta: 0.0
Delta: 0.0
Delta: 3.0
Delta: 3.0
```

The result is the same as before, but now the code is clearer and more intuitive. `yield from` essentially causes the Python interpreter to handle the nested `for` loop and `yield` expression boilerplate for you, which results in better performance. Here, I verify the speedup by using the `timeit` built-in module to run a micro-benchmark:

[Click here to view code image](#)

```

import timeit

def child():
    for i in range(1_000_000):
        yield i

def slow():
    for i in child():
        yield i

def fast():
    yield from child()

baseline = timeit.timeit(
    stmt='for _ in slow(): pass',
    globals=globals(),
    number=50)
print(f'Manual nesting {baseline:.2f}s')
comparison = timeit.timeit(
    stmt='for _ in fast(): pass',
    globals=globals(),
    number=50)
print(f'Composed nesting {comparison:.2f}s')

reduction = -(comparison - baseline) / baseline
print(f'{reduction:.1%} less time')

>>>
Manual nesting 4.02s
Composed nesting 3.47s
13.5% less time

```

If you find yourself composing generators, I strongly encourage you to use `yield from` when possible.

## Things to Remember

- ✦ The `yield from` expression allows you to compose multiple nested generators together into a single combined generator.
- ✦ `yield from` provides better performance than manually iterating nested generators and yielding their outputs.

## Item 34: Avoid Injecting Data into Generators with `send`

yield expressions provide generator functions with a simple way to produce an iterable series of output values (see [Item 30: “Consider Generators Instead of Returning Lists”](#)). However, this channel appears to be unidirectional: There’s no immediately obvious way to simultaneously stream data in and out of a generator as it runs. Having such bidirectional communication could be valuable for a variety of use cases.

For example, say that I’m writing a program to transmit signals using a software-defined radio. Here, I use a function to generate an approximation of a sine wave with a given number of points:

[Click here to view code image](#)

```
import math

def wave(amplitude, steps):
    step_size = 2 * math.pi / steps
    for step in range(steps):
        radians = step * step_size
        fraction = math.sin(radians)
        output = amplitude * fraction
        yield output
```

Now, I can transmit the wave signal at a single specified amplitude by iterating over the wave generator:

[Click here to view code image](#)

```
def transmit(output):
    if output is None:
        print(f'Output is None')
    else:
        print(f'Output: {output:>5.1f}')

def run(it):
    for output in it:
        transmit(output)
```

```
run(wave(3.0, 8))
```

```
>>>
Output:  0.0
Output:  2.1
Output:  3.0
Output:  2.1
```

```
Output: 0.0
Output: -2.1
Output: -3.0
Output: -2.1
```

This works fine for producing basic waveforms, but it can't be used to constantly vary the amplitude of the wave based on a separate input (i.e., as required to broadcast AM radio signals). I need a way to modulate the amplitude on each iteration of the generator.

Python generators support the `send` method, which upgrades `yield` expressions into a two-way channel. The `send` method can be used to provide streaming inputs to a generator at the same time it's yielding outputs. Normally, when iterating a generator, the value of the `yield` expression is `None`:

[Click here to view code image](#)

```
def my_generator():
    received = yield 1
    print(f'received = {received}')

it = iter(my_generator())
output = next(it)          # Get first generator output
print(f'output = {output}')
try:
    next(it)               # Run generator until it exits
except StopIteration:
    pass

>>>
output = 1
received = None
```

When I call the `send` method instead of iterating the generator with a `for` loop or the `next` built-in function, the supplied parameter becomes the value of the `yield` expression when the generator is resumed. However, when the generator first starts, a `yield` expression has not been encountered yet, so the only valid value for calling `send` initially is `None` (any other argument would raise an exception at runtime):

[Click here to view code image](#)

```

it = iter(my_generator())
output = it.send(None) # Get first generator output
print(f'output = {output}')

try:
    it.send('hello!') # Send value into the generator
except StopIteration:
    pass

>>>
output = 1
received = hello!

```

I can take advantage of this behavior in order to modulate the amplitude of the sine wave based on an input signal. First, I need to change the wave generator to save the amplitude returned by the yield expression and use it to calculate the next generated output:

[Click here to view code image](#)

```

def wave_modulating(steps):
    step_size = 2 * math.pi / steps
    amplitude = yield # Receive initial amplitude
    for step in range(steps):
        radians = step * step_size
        fraction = math.sin(radians)
        output = amplitude * fraction
        amplitude = yield output # Receive next amplitude

```

Then, I need to update the run function to stream the modulating amplitude into the wave\_modulating generator on each iteration. The first input to send must be None, since a yield expression would not have occurred within the generator yet:

[Click here to view code image](#)

```

def run_modulating(it):
    amplitudes = [
        None, 7, 7, 7, 2, 2, 2, 2, 10, 10, 10, 10, 10]
    for amplitude in amplitudes:
        output = it.send(amplitude)
        transmit(output)

run_modulating(wave_modulating(12))

```

```
>>>
Output is None
Output: 0.0
Output: 3.5
Output: 6.1
Output: 2.0
Output: 1.7
Output: 1.0
Output: 0.0
Output: -5.0
Output: -8.7
Output: -10.0
Output: -8.7
Output: -5.0
```

This works; it properly varies the output amplitude based on the input signal. The first output is `None`, as expected, because a value for the amplitude wasn't received by the generator until after the initial `yield` expression.

One problem with this code is that it's difficult for new readers to understand: Using `yield` on the right side of an assignment statement isn't intuitive, and it's hard to see the connection between `yield` and `send` without already knowing the details of this advanced generator feature.

Now, imagine that the program's requirements get more complicated. Instead of using a simple sine wave as my carrier, I need to use a complex waveform consisting of multiple signals in sequence. One way to implement this behavior is by composing multiple generators together by using the `yield from` expression (see [Item 33: “Compose Multiple Generators with `yield from`”](#)). Here, I confirm that this works as expected in the simpler case where the amplitude is fixed:

```
def complex_wave():
    yield from wave(7.0, 3)
    yield from wave(2.0, 4)
    yield from wave(10.0, 5)

run(complex_wave())

>>>
Output: 0.0
Output: 6.1
```

```
Output: -6.1
Output: 0.0
Output: 2.0
Output: 0.0
Output: -2.0
Output: 0.0
Output: 9.5
Output: 5.9
Output: -5.9
Output: -9.5
```

Given that the `yield from` expression handles the simpler case, you may expect it to also work properly along with the generator `send` method. Here, I try to use it this way by composing multiple calls to the `wave_modulating` generator together:

[Click here to view code image](#)

```
def complex_wave_modulating():
    yield from wave_modulating(3)
    yield from wave_modulating(4)
    yield from wave_modulating(5)

run_modulating(complex_wave_modulating())
```

```
>>>
Output is None
Output: 0.0
Output: 6.1
Output: -6.1
Output is None
Output: 0.0
Output: 2.0
Output: 0.0
Output: -10.0
Output is None
Output: 0.0
Output: 9.5
Output: 5.9
```

This works to some extent, but the result contains a big surprise: There are many `None` values in the output! Why does this happen? When each `yield from` expression finishes iterating over a nested generator, it moves on to the next one. Each nested generator starts with a bare `yield` expression—one without a value—in order to receive the initial amplitude from a generator

send method call. This causes the parent generator to output a None value when it transitions between child generators.

This means that assumptions about how the `yield from` and `send` features behave individually will be broken if you try to use them together. Although it's possible to work around this None problem by increasing the complexity of the `run_modulating` function, it's not worth the trouble. It's already difficult for new readers of the code to understand how `send` works. This surprising gotcha with `yield from` makes it even worse. My advice is to avoid the `send` method entirely and go with a simpler approach.

The easiest solution is to pass an iterator into the wave function. The iterator should return an input amplitude each time the `next` built-in function is called on it. This arrangement ensures that each generator is progressed in a cascade as inputs and outputs are processed (see [Item 32: “Consider Generator Expressions for Large List Comprehensions”](#) for another example):

[Click here to view code image](#)

```
def wave_cascading(amplitude_it, steps):
    step_size = 2 * math.pi / steps
    for step in range(steps):
        radians = step * step_size
        fraction = math.sin(radians)
        amplitude = next(amplitude_it) # Get next input
        output = amplitude * fraction
        yield output
```

I can pass the same iterator into each of the generator functions that I'm trying to compose together. Iterators are stateful (see [Item 31: “Be Defensive When Iterating Over Arguments”](#)), and thus each of the nested generators picks up where the previous generator left off:

[Click here to view code image](#)

```
def complex_wave_cascading(amplitude_it):
    yield from wave_cascading(amplitude_it, 3)
    yield from wave_cascading(amplitude_it, 4)
    yield from wave_cascading(amplitude_it, 5)
```



Now, I can run the composed generator by simply passing in an iterator from the `amplitudes` list:

[Click here to view code image](#)

```
def run_cascading():
    amplitudes = [7, 7, 7, 2, 2, 2, 2, 10, 10, 10, 10, 10]
    it = complex_wave_cascading(iter(amplitudes))
    for amplitude in amplitudes:
        output = next(it)
        transmit(output)
```

```
run_cascading()
```

```
>>>
```

```
Output: 0.0
Output: 6.1
Output: -6.1
Output: 0.0
Output: 2.0
Output: 0.0
Output: -2.0
Output: 0.0
Output: 9.5
Output: 5.9
Output: -5.9
Output: -9.5
```

The best part about this approach is that the iterator can come from anywhere and could be completely dynamic (e.g., implemented using a generator function). The only downside is that this code assumes that the input generator is completely thread safe, which may not be the case. If you need to cross thread boundaries, `async` functions may be a better fit (see [Item 62: “Mix Threads and Coroutines to Ease the Transition to `asyncio`”](#)).

## Things to Remember

- ✦ The `send` method can be used to inject data into a generator by giving the `yield` expression a value that can be assigned to a variable.
- ✦ Using `send` with `yield` from expressions may cause surprising behavior, such as `None` values appearing at unexpected times in the generator output.

- ✦ Providing an input iterator to a set of composed generators is a better approach than using the `send` method, which should be avoided.

## Item 35: Avoid Causing State Transitions in Generators with `throw`

In addition to `yield` from expressions (see [Item 33: “Compose Multiple Generators with `yield from`”](#)) and the `send` method (see [Item 34: “Avoid Injecting Data into Generators with `send`”](#)), another advanced generator feature is the `throw` method for re-raising `Exception` instances within generator functions. The way `throw` works is simple: When the method is called, the next occurrence of a `yield` expression re-raises the provided `Exception` instance after its output is received instead of continuing normally. Here, I show a simple example of this behavior in action:

[Click here to view code image](#)

```
class MyError(Exception):
    pass

def my_generator():
    yield 1
    yield 2
    yield 3

it = my_generator()
print(next(it)) # Yield 1
print(next(it)) # Yield 2
print(it.throw(MyError('test error')))
```

```
>>>
1
2
Traceback ...
MyError: test error
```

When you call `throw`, the generator function may catch the injected exception with a standard `try/except` compound statement that surrounds the last `yield` expression that was executed (see [Item 65: “Take Advantage of Each Block in `try/except/else/finally`”](#) for more about exception handling):

[Click here to view code image](#)

```
def my_generator():
    yield 1

    try:
        yield 2
    except MyError:
        print('Got MyError!')
    else:
        yield 3

    yield 4

it = my_generator()
print(next(it)) # Yield 1
print(next(it)) # Yield 2
print(it.throw(MyError('test error'))))

>>>
1
2
Got MyError!
4
```

This functionality provides a two-way communication channel between a generator and its caller that can be useful in certain situations (see [Item 34: “Avoid Injecting Data into Generators with send”](#) for another one). For example, imagine that I’m trying to write a program with a timer that supports sporadic resets. Here, I implement this behavior by defining a generator that relies on the `throw` method:

```
class Reset(Exception):
    pass

def timer(period):
    current = period
    while current:
        current -= 1
        try:
            yield current
        except Reset:
            current = period
```

In this code, whenever the `Reset` exception is raised by the `yield` expression, the counter resets itself to its original period.

I can connect this counter reset event to an external input that's polled every second. Then, I can define a `run` function to drive the timer generator, which injects exceptions with `throw` to cause resets, or calls `announce` for each generator output:

[Click here to view code image](#)

```
def check_for_reset():
    # Poll for external event
    ...

def announce(remaining):
    print(f'{remaining} ticks remaining')

def run():
    it = timer(4)
    while True:
        try:
            if check_for_reset():
                current = it.throw(Reset())
            else:
                current = next(it)
        except StopIteration:
            break
        else:
            announce(current)

run()

>>>
3 ticks remaining
2 ticks remaining
1 ticks remaining
3 ticks remaining
2 ticks remaining
3 ticks remaining
2 ticks remaining
1 ticks remaining
0 ticks remaining
```

This code works as expected, but it's much harder to read than necessary. The various levels of nesting required to catch `StopIteration` exceptions or

decide to throw, call next, or announce make the code noisy.

A simpler approach to implementing this functionality is to define a stateful closure (see [Item 38: “Accept Functions Instead of Classes for Simple Interfaces”](#)) using an iterable container object (see [Item 31: “Be Defensive When Iterating Over Arguments”](#)). Here, I redefine the timer generator by using such a class:

[Click here to view code image](#)

```
class Timer:
    def __init__(self, period):
        self.current = period
        self.period = period

    def reset(self):
        self.current = self.period

    def __iter__(self):
        while self.current:
            self.current -= 1
            yield self.current
```

Now, the run method can do a much simpler iteration by using a for statement, and the code is much easier to follow because of the reduction in the levels of nesting:

```
def run():
    timer = Timer(4)
    for current in timer:
        if check_for_reset():
            timer.reset()
        announce(current)
```

```
run()
```

```
>>>
```

```
3 ticks remaining
2 ticks remaining
1 ticks remaining
3 ticks remaining
2 ticks remaining
3 ticks remaining
2 ticks remaining
```

```
1 ticks remaining
0 ticks remaining
```

The output matches the earlier version using `throw`, but this implementation is much easier to understand, especially for new readers of the code. Often, what you’re trying to accomplish by mixing generators and exceptions is better achieved with asynchronous features (see [Item 60: “Achieve Highly Concurrent I/O with Coroutines”](#)). Thus, I suggest that you avoid using `throw` entirely and instead use an iterable class if you need this type of exceptional behavior.

## Things to Remember

- ◆ The `throw` method can be used to re-raise exceptions within generators at the position of the most recently executed `yield` expression.
- ◆ Using `throw` harms readability because it requires additional nesting and boilerplate in order to raise and catch exceptions.
- ◆ A better way to provide exceptional behavior in generators is to use a class that implements the `__iter__` method along with methods to cause exceptional state transitions.

## Item 36: Consider `itertools` for Working with Iterators and Generators

The `itertools` built-in module contains a large number of functions that are useful for organizing and interacting with iterators (see [Item 30: “Consider Generators Instead of Returning Lists”](#) and [Item 31: “Be Defensive When Iterating Over Arguments”](#) for background):

```
import itertools
```

Whenever you find yourself dealing with tricky iteration code, it’s worth looking at the `itertools` documentation again to see if there’s anything in there for you to use (see `help(itertools)`). The following sections describe the most important functions that you should know in three primary categories.

## Linking Iterators Together

The `itertools` built-in module includes a number of functions for linking iterators together.

### *chain*

Use `chain` to combine multiple iterators into a single sequential iterator:

[Click here to view code image](#)

```
it = itertools.chain([1, 2, 3], [4, 5, 6])
print(list(it))

>>>
[1, 2, 3, 4, 5, 6]
```

### *repeat*

Use `repeat` to output a single value forever, or use the second parameter to specify a maximum number of times:

```
it = itertools.repeat('hello', 3)
print(list(it))

>>>
['hello', 'hello', 'hello']
```

### *cycle*

Use `cycle` to repeat an iterator's items forever:

[Click here to view code image](#)

```
it = itertools.cycle([1, 2])
result = [next(it) for _ in range(10)]
print(result)

>>>
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```

### *tee*

Use `tee` to split a single iterator into the number of parallel iterators specified by the second parameter. The memory usage of this function will grow if the iterators don't progress at the same speed since buffering will be required to enqueue the pending items:

[Click here to view code image](#)

```
it1, it2, it3 = itertools.tee(['first', 'second'], 3)
print(list(it1))
print(list(it2))
print(list(it3))

>>>
['first', 'second']
['first', 'second']
['first', 'second']
```

## *zip\_longest*

This variant of the `zip` built-in function (see [Item 8: “Use `zip` to Process Iterators in Parallel](#)”) returns a placeholder value when an iterator is exhausted, which may happen if iterators have different lengths:

[Click here to view code image](#)

```
keys = ['one', 'two', 'three']
values = [1, 2]

normal = list(zip(keys, values))
print('zip: ', normal)

it = itertools.zip_longest(keys, values, fillvalue='nope')
longest = list(it)
print('zip_longest:', longest)

>>>
zip:      [('one', 1), ('two', 2)]
zip_longest: [('one', 1), ('two', 2), ('three', 'nope')]
```

## **Filtering Items from an Iterator**

The `itertools` built-in module includes a number of functions for filtering items from an iterator.



## *islice*

Use `islice` to slice an iterator by numerical indexes without copying. You can specify the end, start and end, or start, end, and step sizes, and the behavior is similar to that of standard sequence slicing and striding (see [Item 11: “Know How to Slice Sequences”](#) and [Item 12: “Avoid Striding and Slicing in a Single Expression”](#)):

[Click here to view code image](#)

```
values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

first_five = itertools.islice(values, 5)
print('First five: ', list(first_five))

middle_odds = itertools.islice(values, 2, 8, 2)
print('Middle odds:', list(middle_odds))

>>>
First five: [1, 2, 3, 4, 5]
Middle odds: [3, 5, 7]
```

## *takewhile*

`takewhile` returns items from an iterator until a predicate function returns `False` for an item:

[Click here to view code image](#)

```
values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
less_than_seven = lambda x: x < 7
it = itertools.takewhile(less_than_seven, values)
print(list(it))

>>>
[1, 2, 3, 4, 5, 6]
```

## *dropwhile*

`dropwhile`, which is the opposite of `takewhile`, skips items from an iterator until the predicate function returns `True` for the first time:

[Click here to view code image](#)

```
values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
less_than_seven = lambda x: x < 7
it = itertools.dropwhile(less_than_seven, values)
print(list(it))
```

```
>>>
[7, 8, 9, 10]
```

## *filterfalse*

`filterfalse`, which is the opposite of the `filter` built-in function, returns all items from an iterator where a predicate function returns `False`:

[Click here to view code image](#)

```
values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
evens = lambda x: x % 2 == 0

filter_result = filter(evens, values)
print('Filter:      ', list(filter_result))

filter_false_result = itertools.filterfalse(evens, values)
print('Filter false:', list(filter_false_result))

>>>
Filter:      [2, 4, 6, 8, 10]
Filter false: [1, 3, 5, 7, 9]
```

## Producing Combinations of Items from Iterators

The `itertools` built-in module includes a number of functions for producing combinations of items from iterators.

### *accumulate*

`accumulate` folds an item from the iterator into a running value by applying a function that takes two parameters. It outputs the current accumulated result for each input value:

[Click here to view code image](#)

```
values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sum_reduce = itertools.accumulate(values)
print('Sum:      ', list(sum_reduce))
```

```
def sum_modulo_20(first, second):
    output = first + second
    return output % 20

modulo_reduce = itertools.accumulate(values, sum_modulo_20)
print('Modulo:', list(modulo_reduce))

>>>
Sum:      [1, 3, 6, 10, 15, 21, 28, 36, 45, 55]
Modulo:   [1, 3, 6, 10, 15, 1, 8, 16, 5, 15]
```

This is essentially the same as the `reduce` function from the `functools` built-in module, but with outputs yielded one step at a time. By default it sums the inputs if no binary function is specified.

## *product*

`product` returns the Cartesian product of items from one or more iterators, which is a nice alternative to using deeply nested list comprehensions (see [Item 28: “Avoid More Than Two Control Subexpressions in Comprehensions”](#) for why to avoid those):

[Click here to view code image](#)

```
single = itertools.product([1, 2], repeat=2)
print('Single: ', list(single))

multiple = itertools.product([1, 2], ['a', 'b'])
print('Multiple:', list(multiple))

>>>
Single:  [(1, 1), (1, 2), (2, 1), (2, 2)]
Multiple: [(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')]
```

## *permutations*

`permutations` returns the unique ordered permutations of length  $N$  with items from an iterator:

[Click here to view code image](#)

```
it = itertools.permutations([1, 2, 3, 4], 2)
print(list(it))
```

```
>>>
[(1, 2),
 (1, 3),
 (1, 4),
 (2, 1),
 (2, 3),
 (2, 4),
 (3, 1),
 (3, 2),
 (3, 4),
 (4, 1),
 (4, 2),
 (4, 3)]
```

## *combinations*

`combinations` returns the unordered combinations of length  $N$  with unrepeated items from an iterator:

[Click here to view code image](#)

```
it = itertools.combinations([1, 2, 3, 4], 2)
print(list(it))
```

```
>>>
[(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]
```

## *combinations\_with\_replacement*

`combinations_with_replacement` is the same as `combinations`, but repeated values are allowed:

[Click here to view code image](#)

```
it = itertools.combinations_with_replacement([1, 2, 3, 4], 2)
print(list(it))
```

```
>>>
[(1, 1),
 (1, 2),
 (1, 3),
 (1, 4),
 (2, 2),
 (2, 3),
 (2, 4),
```

```
(3, 3),  
(3, 4),  
(4, 4)]
```

## Things to Remember

- ◆ The `itertools` functions fall into three main categories for working with iterators and generators: linking iterators together, filtering items they output, and producing combinations of items.
- ◆ There are more advanced functions, additional parameters, and useful recipes available in the documentation at `help(itertools)`.

## 5. Classes and Interfaces

As an object-oriented programming language, Python supports a full range of features, such as inheritance, polymorphism, and encapsulation. Getting things done in Python often requires writing new classes and defining how they interact through their interfaces and hierarchies.

Python's classes and inheritance make it easy to express a program's intended behaviors with objects. They allow you to improve and expand functionality over time. They provide flexibility in an environment of changing requirements. Knowing how to use them well enables you to write maintainable code.

### Item 37: Compose Classes Instead of Nesting Many Levels of Built-in Types

Python's built-in dictionary type is wonderful for maintaining dynamic internal state over the lifetime of an object. By *dynamic*, I mean situations in which you need to do bookkeeping for an unexpected set of identifiers. For example, say that I want to record the grades of a set of students whose names aren't known in advance. I can define a class to store the names in a dictionary instead of using a predefined attribute for each student:

[Click here to view code image](#)

```
class SimpleGradebook:
    def __init__(self):
        self._grades = {}

    def add_student(self, name):
        self._grades[name] = []

    def report_grade(self, name, score):
        self._grades[name].append(score)

    def average_grade(self, name):
        grades = self._grades[name]
        return sum(grades) / len(grades)
```

Using the class is simple:

[Click here to view code image](#)

```
book = SimpleGradebook()
book.add_student('Isaac Newton')
book.report_grade('Isaac Newton', 90)
book.report_grade('Isaac Newton', 95)
book.report_grade('Isaac Newton', 85)

print(book.average_grade('Isaac Newton'))

>>>
90.0
```

Dictionaries and their related built-in types are so easy to use that there's a danger of overextending them to write brittle code. For example, say that I want to extend the `SimpleGradebook` class to keep a list of grades by subject, not just overall. I can do this by changing the `_grades` dictionary to map student names (its keys) to yet another dictionary (its values). The innermost dictionary will map subjects (its keys) to a list of grades (its values). Here, I do this by using a `defaultdict` instance for the inner dictionary to handle missing subjects (see [Item 17: “Prefer defaultdict Over setdefault to Handle Missing Items in Internal State”](#) for background):

[Click here to view code image](#)

```
from collections import defaultdict

class BySubjectGradebook:
    def __init__(self):
        self._grades = {} # Outer dict

    def add_student(self, name):
        self._grades[name] = defaultdict(list) # Inner dict
```

This seems straightforward enough. The `report_grade` and `average_grade` methods gain quite a bit of complexity to deal with the multilevel dictionary, but it's seemingly manageable:

[Click here to view code image](#)

```

def report_grade(self, name, subject, grade):
    by_subject = self._grades[name]
    grade_list = by_subject[subject]
    grade_list.append(grade)

def average_grade(self, name):
    by_subject = self._grades[name]
    total, count = 0, 0
    for grades in by_subject.values():
        total += sum(grades)
        count += len(grades)
    return total / count

```

Using the class remains simple:

[Click here to view code image](#)

```

book = BySubjectGradebook()
book.add_student('Albert Einstein')
book.report_grade('Albert Einstein', 'Math', 75)
book.report_grade('Albert Einstein', 'Math', 65)
book.report_grade('Albert Einstein', 'Gym', 90)
book.report_grade('Albert Einstein', 'Gym', 95)
print(book.average_grade('Albert Einstein'))

>>>
81.25

```

Now, imagine that the requirements change again. I also want to track the weight of each score toward the overall grade in the class so that midterm and final exams are more important than pop quizzes. One way to implement this feature is to change the innermost dictionary; instead of mapping subjects (its keys) to a list of grades (its values), I can use the tuple of (score, weight) in the values list:

[Click here to view code image](#)

```

class WeightedGradebook:
    def __init__(self):
        self._grades = {}

    def add_student(self, name):
        self._grades[name] = defaultdict(list)

    def report_grade(self, name, subject, score, weight):
        by_subject = self._grades[name]

```



```
grade_list = by_subject[subject]
grade_list.append((score, weight))
```

Although the changes to `report_grade` seem simple—just make the grade list store tuple instances—the `average_grade` method now has a loop within a loop and is difficult to read:

[Click here to view code image](#)

```
def average_grade(self, name):
    by_subject = self._grades[name]

    score_sum, score_count = 0, 0
    for subject, scores in by_subject.items():
        subject_avg, total_weight = 0, 0
        for score, weight in scores:
            subject_avg += score * weight
            total_weight += weight

    score_sum += subject_avg / total_weight
    score_count += 1

    return score_sum / score_count
```

Using the class has also gotten more difficult. It's unclear what all of the numbers in the positional arguments mean:

[Click here to view code image](#)

```
book = WeightedGradebook()
book.add_student('Albert Einstein')
book.report_grade('Albert Einstein', 'Math', 75, 0.05)
book.report_grade('Albert Einstein', 'Math', 65, 0.15)
book.report_grade('Albert Einstein', 'Math', 70, 0.80)
book.report_grade('Albert Einstein', 'Gym', 100, 0.40)
book.report_grade('Albert Einstein', 'Gym', 85, 0.60)
print(book.average_grade('Albert Einstein'))

>>>
80.25
```

When you see complexity like this, it's time to make the leap from built-in types like dictionaries, tuples, sets, and lists to a hierarchy of classes.

In the grades example, at first I didn't know I'd need to support weighted grades, so the complexity of creating classes seemed unwarranted. Python's

built-in dictionary and tuple types made it easy to keep going, adding layer after layer to the internal bookkeeping. But you should avoid doing this for more than one level of nesting; using dictionaries that contain dictionaries makes your code hard to read by other programmers and sets you up for a maintenance nightmare.

As soon as you realize that your bookkeeping is getting complicated, break it all out into classes. You can then provide well-defined interfaces that better encapsulate your data. This approach also enables you to create a layer of abstraction between your interfaces and your concrete implementations.

## Refactoring to Classes

There are many approaches to refactoring (see [Item 89: “Consider warnings to Refactor and Migrate Usage”](#) for another). In this case, I can start moving to classes at the bottom of the dependency tree: a single grade. A class seems too heavyweight for such simple information. A tuple, though, seems appropriate because grades are immutable. Here, I use the tuple of (score, weight) to track grades in a list:

[Click here to view code image](#)

```
grades = []
grades.append((95, 0.45))
grades.append((85, 0.55))
total = sum(score * weight for score, weight in grades)
total_weight = sum(weight for _, weight in grades)
average_grade = total / total_weight
```

I used `_` (the underscore variable name, a Python convention for unused variables) to capture the first entry in each grade’s tuple and ignore it when calculating the `total_weight`.

The problem with this code is that tuple instances are positional. For example, if I want to associate more information with a grade, such as a set of notes from the teacher, I need to rewrite every usage of the two-tuple to be aware that there are now three items present instead of two, which means I need to use `_` further to ignore certain indexes:

[Click here to view code image](#)

```
grades = []
grades.append((95, 0.45, 'Great job'))
grades.append((85, 0.55, 'Better next time'))
total = sum(score * weight for score, weight, _ in grades)
total_weight = sum(weight for _, weight, _ in grades)
average_grade = total / total_weight
```

This pattern of extending tuples longer and longer is similar to deepening layers of dictionaries. As soon as you find yourself going longer than a two-tuple, it's time to consider another approach.

The `namedtuple` type in the `collections` built-in module does exactly what I need in this case: It lets me easily define tiny, immutable data classes:

[Click here to view code image](#)

```
from collections import namedtuple

Grade = namedtuple('Grade', ('score', 'weight'))
```

These classes can be constructed with positional or keyword arguments. The fields are accessible with named attributes. Having named attributes makes it easy to move from a `namedtuple` to a class later if the requirements change again and I need to, say, support mutability or behaviors in the simple data containers.

### Limitations of `namedtuple`

Although `namedtuple` is useful in many circumstances, it's important to understand when it can do more harm than good:

- You can't specify default argument values for `namedtuple` classes. This makes them unwieldy when your data may have many optional properties. If you find yourself using more than a handful of attributes, using the built-in `dataclasses` module may be a better choice.
- The attribute values of `namedtuple` instances are still accessible using numerical indexes and iteration. Especially in externalized

APIs, this can lead to unintentional usage that makes it harder to move to a real class later. If you're not in control of all of the usage of your `namedtuple` instances, it's better to explicitly define a new class.

Next, I can write a class to represent a single subject that contains a set of grades:

[Click here to view code image](#)

```
class Subject:
    def __init__(self):
        self._grades = []

    def report_grade(self, score, weight):
        self._grades.append(Grade(score, weight))

    def average_grade(self):
        total, total_weight = 0, 0
        for grade in self._grades:
            total += grade.score * grade.weight
            total_weight += grade.weight
        return total / total_weight
```

Then, I write a class to represent a set of subjects that are being studied by a single student:

[Click here to view code image](#)

```
class Student:
    def __init__(self):
        self._subjects = defaultdict(Subject)

    def get_subject(self, name):
        return self._subjects[name]

    def average_grade(self):
        total, count = 0, 0
        for subject in self._subjects.values():
            total += subject.average_grade()
            count += 1
        return total / count
```

Finally, I'd write a container for all of the students, keyed dynamically by their names:

[Click here to view code image](#)

```
class Gradebook:
    def __init__(self):
        self._students = defaultdict(Student)

    def get_student(self, name):
        return self._students[name]
```

The line count of these classes is almost double the previous implementation's size. But this code is much easier to read. The example driving the classes is also more clear and extensible:

[Click here to view code image](#)

```
book = Gradebook()
albert = book.get_student('Albert Einstein')
math = albert.get_subject('Math')
math.report_grade(75, 0.05)
math.report_grade(65, 0.15)
math.report_grade(70, 0.80)
gym = albert.get_subject('Gym')
gym.report_grade(100, 0.40)
gym.report_grade(85, 0.60)
print(albert.average_grade())

>>>
80.25
```

It would also be possible to write backward-compatible methods to help migrate usage of the old API style to the new hierarchy of objects.

## Things to Remember

- ✦ Avoid making dictionaries with values that are dictionaries, long tuples, or complex nestings of other built-in types.
- ✦ Use `namedtuple` for lightweight, immutable data containers before you need the flexibility of a full class.

- ✦ Move your bookkeeping code to using multiple classes when your internal state dictionaries get complicated.

## Item 38: Accept Functions Instead of Classes for Simple Interfaces

Many of Python’s built-in APIs allow you to customize behavior by passing in a function. These *hooks* are used by APIs to call back your code while they execute. For example, the `list` type’s `sort` method takes an optional `key` argument that’s used to determine each index’s value for sorting (see [Item 14: “Sort by Complex Criteria Using the `key` Parameter”](#) for details). Here, I sort a `list` of names based on their lengths by providing the `len` built-in function as the `key` hook:

[Click here to view code image](#)

```
names = ['Socrates', 'Archimedes', 'Plato', 'Aristotle']
names.sort(key=len)
print(names)

>>>
['Plato', 'Socrates', 'Aristotle', 'Archimedes']
```

In other languages, you might expect hooks to be defined by an abstract class. In Python, many hooks are just stateless functions with well-defined arguments and return values. Functions are ideal for hooks because they are easier to describe and simpler to define than classes. Functions work as hooks because Python has *first-class* functions: Functions and methods can be passed around and referenced like any other value in the language.

For example, say that I want to customize the behavior of the `defaultdict` class (see [Item 17: “Prefer `defaultdict` Over `setdefault` to Handle Missing Items in Internal State”](#) for background). This data structure allows you to supply a function that will be called with no arguments each time a missing key is accessed. The function must return the default value that the missing key should have in the dictionary. Here, I define a hook that logs each time a key is missing and returns `0` for the default value:

```
def log_missing():
    print('Key added')
    return 0
```

Given an initial dictionary and a set of desired increments, I can cause the `log_missing` function to run and print twice (for 'red' and 'orange'):

[Click here to view code image](#)

```
from collections import defaultdict

current = {'green': 12, 'blue': 3}
increments = [
    ('red', 5),
    ('blue', 17),
    ('orange', 9),
]
result = defaultdict(log_missing, current)
print('Before:', dict(result))
for key, amount in increments:
    result[key] += amount
print('After: ', dict(result))
```

```
>>>
Before: {'green': 12, 'blue': 3}
Key added
Key added
After: {'green': 12, 'blue': 20, 'red': 5, 'orange': 9}
```

Supplying functions like `log_missing` makes APIs easy to build and test because it separates side effects from deterministic behavior. For example, say I now want the default value hook passed to `defaultdict` to count the total number of keys that were missing. One way to achieve this is by using a stateful closure (see [Item 21: “Know How Closures Interact with Variable Scope”](#) for details). Here, I define a helper function that uses such a closure as the default value hook:

[Click here to view code image](#)

```
def increment_with_report(current, increments):
    added_count = 0

    def missing():
        nonlocal added_count # Stateful closure
        added_count += 1
```

```

    return 0

    result = defaultdict(missing, current)
    for key, amount in increments:
        result[key] += amount
    return result, added_count

```

Running this function produces the expected result (2), even though the defaultdict has no idea that the missing hook maintains state. Another benefit of accepting simple functions for interfaces is that it's easy to add functionality later by hiding state in a closure:

[Click here to view code image](#)

```

result, count = increment_with_report(current, increments)
assert count == 2

```

The problem with defining a closure for stateful hooks is that it's harder to read than the stateless function example. Another approach is to define a small class that encapsulates the state you want to track:

```

class CountMissing:
    def __init__(self):
        self.added = 0

    def missing(self):
        self.added += 1
        return 0

```

In other languages, you might expect that now defaultdict would have to be modified to accommodate the interface of CountMissing. But in Python, thanks to first-class functions, you can reference the CountMissing.missing method directly on an object and pass it to defaultdict as the default value hook. It's trivial to have an object instance's method satisfy a function interface:

[Click here to view code image](#)

```

counter = CountMissing()
result = defaultdict(counter.missing, current) # Method ref
for key, amount in increments:
    result[key] += amount
assert counter.added == 2

```



Using a helper class like this to provide the behavior of a stateful closure is clearer than using the `increment_with_report` function, as above. However, in isolation, it's still not immediately obvious what the purpose of the `CountMissing` class is. Who constructs a `CountMissing` object? Who calls the missing method? Will the class need other public methods to be added in the future? Until you see its usage with `defaultdict`, the class is a mystery.

To clarify this situation, Python allows classes to define the `__call__` special method. `__call__` allows an object to be called just like a function. It also causes the `callable` built-in function to return `True` for such an instance, just like a normal function or method. All objects that can be executed in this manner are referred to as *callables*:

```
class BetterCountMissing:
    def __init__(self):
        self.added = 0

    def __call__(self):
        self.added += 1
        return 0

counter = BetterCountMissing()
assert counter() == 0
assert callable(counter)
```

Here, I use a `BetterCountMissing` instance as the default value hook for a `defaultdict` to track the number of missing keys that were added:

[Click here to view code image](#)

```
counter = BetterCountMissing()
result = defaultdict(counter, current) # Relies on __call__
for key, amount in increments:
    result[key] += amount
assert counter.added == 2
```

This is much clearer than the `CountMissing.missing` example. The `__call__` method indicates that a class's instances will be used somewhere a function argument would also be suitable (like API hooks). It directs new readers of the code to the entry point that's responsible for the class's primary behavior. It provides a strong hint that the goal of the class is to act as a stateful closure.

Best of all, `defaultdict` still has no view into what's going on when you use `__call__`. All that `defaultdict` requires is a function for the default value hook. Python provides many different ways to satisfy a simple function interface, and you can choose the one that works best for what you need to accomplish.

## Things to Remember

- ✦ Instead of defining and instantiating classes, you can often simply use functions for simple interfaces between components in Python.
- ✦ References to functions and methods in Python are first class, meaning they can be used in expressions (like any other type).
- ✦ The `__call__` special method enables instances of a class to be called like plain Python functions.
- ✦ When you need a function to maintain state, consider defining a class that provides the `__call__` method instead of defining a stateful closure.

## Item 39: Use `@classmethod` Polymorphism to Construct Objects Generically

In Python, not only do objects support polymorphism, but classes do as well. What does that mean, and what is it good for?

Polymorphism enables multiple classes in a hierarchy to implement their own unique versions of a method. This means that many classes can fulfill the same interface or abstract base class while providing different functionality (see [Item 43: “Inherit from `collections.abc` for Custom Container Types](#)”).

For example, say that I'm writing a MapReduce implementation, and I want a common class to represent the input data. Here, I define such a class with a `read` method that must be defined by subclasses:

```
class InputData:
    def read(self):
```

```
raise NotImplementedError
```

I also have a concrete subclass of `InputData` that reads data from a file on disk:

```
class PathInputData(InputData):
    def __init__(self, path):
        super().__init__()
        self.path = path

    def read(self):
        with open(self.path) as f:
            return f.read()
```

I could have any number of `InputData` subclasses, like `PathInputData`, and each of them could implement the standard interface for `read` to return the data to process. Other `InputData` subclasses could read from the network, decompress data transparently, and so on.

I'd want a similar abstract interface for the MapReduce worker that consumes the input data in a standard way:

[Click here to view code image](#)

```
class Worker:
    def __init__(self, input_data):
        self.input_data = input_data
        self.result = None

    def map(self):
        raise NotImplementedError

    def reduce(self, other):
        raise NotImplementedError
```

Here, I define a concrete subclass of worker to implement the specific MapReduce function I want to apply—a simple newline counter:

[Click here to view code image](#)

```
class LineCountWorker(Worker):
    def map(self):
        data = self.input_data.read()
        self.result = data.count('\n')
    def reduce(self, other):
        self.result += other.result
```

It may look like this implementation is going great, but I've reached the biggest hurdle in all of this. What connects all of these pieces? I have a nice set of classes with reasonable interfaces and abstractions, but that's only useful once the objects are constructed. What's responsible for building the objects and orchestrating the MapReduce?

The simplest approach is to manually build and connect the objects with some helper functions. Here, I list the contents of a directory and construct a PathInputData instance for each file it contains:

[Click here to view code image](#)

```
import os

def generate_inputs(data_dir):
    for name in os.listdir(data_dir):
        yield PathInputData(os.path.join(data_dir, name))
```

Next, I create the LineCountWorker instances by using the InputData instances returned by generate\_inputs:

[Click here to view code image](#)

```
def create_workers(input_list):
    workers = []
    for input_data in input_list:
        workers.append(LineCountWorker(input_data))
    return workers
```

I execute these worker instances by fanning out the map step to multiple threads (see [Item 53: “Use Threads for Blocking I/O, Avoid for Parallelism”](#) for background). Then, I call reduce repeatedly to combine the results into one final value:

[Click here to view code image](#)

```
from threading import Thread

def execute(workers):
    threads = [Thread(target=w.map) for w in workers]
    for thread in threads: thread.start()
    for thread in threads: thread.join()

    first, *rest = workers
```

```
for worker in rest:
    first.reduce(worker)
return first.result
```

Finally, I connect all the pieces together in a function to run each step:

[Click here to view code image](#)

```
def mapreduce(data_dir):
    inputs = generate_inputs(data_dir)
    workers = create_workers(inputs)
    return execute(workers)
```

Running this function on a set of test input files works great:

[Click here to view code image](#)

```
import os
import random

def write_test_files(tmpdir):
    os.makedirs(tmpdir)
    for i in range(100):
        with open(os.path.join(tmpdir, str(i)), 'w') as f:
            f.write('\n' * random.randint(0, 100))

tmpdir = 'test_inputs'
write_test_files(tmpdir)

result = mapreduce(tmpdir)
print(f'There are {result} lines')

>>>
There are 4360 lines
```

What's the problem? The huge issue is that the mapreduce function is not generic at all. If I wanted to write another `InputData` or `Worker` subclass, I would also have to rewrite the `generate_inputs`, `create_workers`, and `mapreduce` functions to match.

This problem boils down to needing a generic way to construct objects. In other languages, you'd solve this problem with constructor polymorphism, requiring that each `InputData` subclass provides a special constructor that can be used generically by the helper methods that orchestrate the MapReduce (similar to the factory pattern). The trouble is that Python only

allows for the single constructor method `__init__`. It's unreasonable to require every `InputData` subclass to have a compatible constructor.

The best way to solve this problem is with *class method* polymorphism. This is exactly like the instance method polymorphism I used for `InputData.read`, except that it's for whole classes instead of their constructed objects.

Let me apply this idea to the MapReduce classes. Here, I extend the `InputData` class with a generic `@classmethod` that's responsible for creating new `InputData` instances using a common interface:

```
class GenericInputData:
    def read(self):
        raise NotImplementedError

    @classmethod
    def generate_inputs(cls, config):
        raise NotImplementedError
```

I have `generate_inputs` take a dictionary with a set of configuration parameters that the `GenericInputData` concrete subclass needs to interpret. Here, I use the `config` to find the directory to list for input files:

[Click here to view code image](#)

```
class PathInputData(GenericInputData):
    ...

    @classmethod
    def generate_inputs(cls, config):
        data_dir = config['data_dir']
        for name in os.listdir(data_dir):
            yield cls(os.path.join(data_dir, name))
```

Similarly, I can make the `create_workers` helper part of the `GenericWorker` class. Here, I use the `input_class` parameter, which must be a subclass of `GenericInputData`, to generate the necessary inputs. I construct instances of the `GenericWorker` concrete subclass by using `cls()` as a generic constructor:

[Click here to view code image](#)

```

class GenericWorker:
    def __init__(self, input_data):
        self.input_data = input_data
        self.result = None

    def map(self):
        raise NotImplementedError

    def reduce(self, other):
        raise NotImplementedError

    @classmethod
    def create_workers(cls, input_class, config):
        workers = []
        for input_data in input_class.generate_inputs(config):
            workers.append(cls(input_data))
        return workers

```

Note that the call to `input_class.generate_inputs` above is the class polymorphism that I'm trying to show. You can also see how `create_workers` calling `cls()` provides an alternative way to construct `GenericWorker` objects besides using the `__init__` method directly.

The effect on my concrete `GenericWorker` subclass is nothing more than changing its parent class:

[Click here to view code image](#)

```

class LineCountWorker(GenericWorker):
    ...

```

Finally, I can rewrite the `mapreduce` function to be completely generic by calling `create_workers`:

[Click here to view code image](#)

```

def mapreduce(worker_class, input_class, config):
    workers = worker_class.create_workers(input_class, config)
    return execute(workers)

```

Running the new worker on a set of test files produces the same result as the old implementation. The difference is that the `mapreduce` function requires more parameters so that it can operate generically:

[Click here to view code image](#)

```
config = {'data_dir': tmpdir}
result = mapreduce(LineCountWorker, PathInputData, config)
print(f'There are {result} lines')

>>>
There are 4360 lines
```

Now, I can write other `GenericInputData` and `GenericWorker` subclasses as I wish, without having to rewrite any of the glue code.

## Things to Remember

- ✦ Python only supports a single constructor per class: the `__init__` method.
- ✦ Use `@classmethod` to define alternative constructors for your classes.
- ✦ Use class method polymorphism to provide generic ways to build and connect many concrete subclasses.

## Item 40: Initialize Parent Classes with `super`

The old, simple way to initialize a parent class from a child class is to directly call the parent class's `__init__` method with the child instance:

```
class MyBaseClass:
    def __init__(self, value):
        self.value = value
class MyChildClass(MyBaseClass):
    def __init__(self):
        MyBaseClass.__init__(self, 5)
```

This approach works fine for basic class hierarchies but breaks in many cases.

If a class is affected by multiple inheritance (something to avoid in general; see [Item 41: “Consider Composing Functionality with Mix-in Classes”](#)), calling the superclasses' `__init__` methods directly can lead to unpredictable behavior.

One problem is that the `__init__` call order isn't specified across all subclasses. For example, here I define two parent classes that operate on the



instance's value field:

```
class TimesTwo:
    def __init__(self):
        self.value *= 2

class PlusFive:
    def __init__(self):
        self.value += 5
```

This class defines its parent classes in one ordering:

[Click here to view code image](#)

```
class OneWay(MyBaseClass, TimesTwo, PlusFive):
    def __init__(self, value):
        MyBaseClass.__init__(self, value)
        TimesTwo.__init__(self)
        PlusFive.__init__(self)
```

And constructing it produces a result that matches the parent class ordering:

[Click here to view code image](#)

```
foo = OneWay(5)
print('First ordering value is (5 * 2) + 5 =', foo.value)

>>>
First ordering value is (5 * 2) + 5 = 15
```

Here's another class that defines the same parent classes but in a different ordering (PlusFive followed by TimesTwo instead of the other way around):

[Click here to view code image](#)

```
class AnotherWay(MyBaseClass, PlusFive, TimesTwo):
    def __init__(self, value):
        MyBaseClass.__init__(self, value)
        TimesTwo.__init__(self)
        PlusFive.__init__(self)
```

However, I left the calls to the parent class constructors—`PlusFive.__init__` and `TimesTwo.__init__`—in the same order as before, which means this class's behavior doesn't match the order of the parent classes in its definition. The conflict here between the inheritance base

classes and the `__init__` calls is hard to spot, which makes this especially difficult for new readers of the code to understand:

[Click here to view code image](#)

```
bar = AnotherWay(5)
print('Second ordering value is', bar.value)

>>>
Second ordering value is 15
```

Another problem occurs with diamond inheritance. Diamond inheritance happens when a subclass inherits from two separate classes that have the same superclass somewhere in the hierarchy. Diamond inheritance causes the common superclass's `__init__` method to run multiple times, causing unexpected behavior. For example, here I define two child classes that inherit from `MyBaseClass`:

[Click here to view code image](#)

```
class TimesSeven(MyBaseClass):
    def __init__(self, value):
        MyBaseClass.__init__(self, value)
        self.value *= 7

class PlusNine(MyBaseClass):
    def __init__(self, value):
        MyBaseClass.__init__(self, value)
        self.value += 9
```

Then, I define a child class that inherits from both of these classes, making `MyBaseClass` the top of the diamond:

[Click here to view code image](#)

```
class ThisWay(TimesSeven, PlusNine):
    def __init__(self, value):
        TimesSeven.__init__(self, value)
        PlusNine.__init__(self, value)

foo = ThisWay(5)
print('Should be (5 * 7) + 9 = 44 but is', foo.value)

>>>
Should be (5 * 7) + 9 = 44 but is 14
```

The call to the second parent class's constructor, `PlusNine.__init__`, causes `self.value` to be reset back to 5 when `MyBaseClass.__init__` gets called a second time. That results in the calculation of `self.value` to be  $5 + 9 = 14$ , completely ignoring the effect of the `TimesSeven.__init__` constructor. This behavior is surprising and can be very difficult to debug in more complex cases.

To solve these problems, Python has the `super` built-in function and standard method resolution order (MRO). `super` ensures that common superclasses in diamond hierarchies are run only once (for another example, see [Item 48: “Validate Subclasses with `\_\_init\_subclass\_\_`”](#)). The MRO defines the ordering in which superclasses are initialized, following an algorithm called *C3 linearization*.

Here, I create a diamond-shaped class hierarchy again, but this time I use `super` to initialize the parent class:

```
class TimesSevenCorrect(MyBaseClass):
    def __init__(self, value):
        super().__init__(value)
        self.value *= 7
```

```
class PlusNineCorrect(MyBaseClass):
    def __init__(self, value):
        super().__init__(value)
        self.value += 9
```

Now, the top part of the diamond, `MyBaseClass.__init__`, is run only a single time. The other parent classes are run in the order specified in the class statement:

[Click here to view code image](#)

```
class GoodWay(TimesSevenCorrect, PlusNineCorrect):
    def __init__(self, value):
        super().__init__(value)

foo = GoodWay(5)
print('Should be 7 * (5 + 9) = 98 and is', foo.value)

>>>
Should be 7 * (5 + 9) = 98 and is 98
```

This order may seem backward at first. Shouldn't TimesSevenCorrect.\_\_init\_\_ have run first? Shouldn't the result be  $(5 * 7) + 9 = 44$ ? The answer is no. This ordering matches what the MRO defines for this class. The MRO ordering is available on a class method called mro:

[Click here to view code image](#)

```
mro_str = '\n'.join(repr(cls) for cls in GoodWay.mro())
print(mro_str)
```

```
>>>
<class '__main__.GoodWay'>
<class '__main__.TimesSevenCorrect'>
<class '__main__.PlusNineCorrect'>
<class '__main__.MyBaseClass'>
<class 'object'>
```

When I call GoodWay(5), it in turn calls TimesSevenCorrect.\_\_init\_\_, which calls PlusNineCorrect.\_\_init\_\_, which calls MyBaseClass.\_\_init\_\_. Once this reaches the top of the diamond, all of the initialization methods actually do their work in the opposite order from how their \_\_init\_\_ functions were called. MyBaseClass.\_\_init\_\_ assigns value to 5.

PlusNineCorrect.\_\_init\_\_ adds 9 to make value equal 14.

TimesSevenCorrect.\_\_init\_\_ multiplies it by 7 to make value equal 98.

Besides making multiple inheritance robust, the call to super(). \_\_init\_\_ is also much more maintainable than calling MyBaseClass.\_\_init\_\_ directly from within the subclasses. I could later rename MyBaseClass to something else or have TimesSevenCorrect and PlusNineCorrect inherit from another superclass without having to update their \_\_init\_\_ methods to match.

The super function can also be called with two parameters: first the type of the class whose MRO parent view you're trying to access, and then the instance on which to access that view. Using these optional parameters within the constructor looks like this:

[Click here to view code image](#)

```
class ExplicitTrisect(MyBaseClass):
    def __init__(self, value):
```

```
super(ExplicitTrisect, self).__init__(value)
self.value /= 3
```

However, these parameters are not required for object instance initialization. Python's compiler automatically provides the correct parameters (`__class__` and `self`) for you when `super` is called with zero arguments within a class definition. This means all three of these usages are equivalent:

[Click here to view code image](#)

```
class AutomaticTrisect(MyBaseClass):
    def __init__(self, value):
        super(__class__, self).__init__(value)
        self.value /= 3
```

```
class ImplicitTrisect(MyBaseClass):
    def __init__(self, value):
        super().__init__(value)
        self.value /= 3
```

```
assert ExplicitTrisect(9).value == 3
assert AutomaticTrisect(9).value == 3
assert ImplicitTrisect(9).value == 3
```

The only time you should provide parameters to `super` is in situations where you need to access the specific functionality of a superclass's implementation from a child class (e.g., to wrap or reuse functionality).

## Things to Remember

- ✦ Python's standard method resolution order (MRO) solves the problems of superclass initialization order and diamond inheritance.
- ✦ Use the `super` built-in function with zero arguments to initialize parent classes.

## Item 41: Consider Composing Functionality with Mix-in Classes

Python is an object-oriented language with built-in facilities for making multiple inheritance tractable (see [Item 40: “Initialize Parent Classes with `super`”](#)). However, it’s better to avoid multiple inheritance altogether.

If you find yourself desiring the convenience and encapsulation that come with multiple inheritance, but want to avoid the potential headaches, consider writing a *mix-in* instead. A mix-in is a class that defines only a small set of additional methods for its child classes to provide. Mix-in classes don’t define their own instance attributes nor require their `__init__` constructor to be called.

Writing mix-ins is easy because Python makes it trivial to inspect the current state of any object, regardless of its type. Dynamic inspection means you can write generic functionality just once, in a mix-in, and it can then be applied to many other classes. Mix-ins can be composed and layered to minimize repetitive code and maximize reuse.

For example, say I want the ability to convert a Python object from its in-memory representation to a dictionary that’s ready for serialization. Why not write this functionality generically so I can use it with all my classes?

Here, I define an example mix-in that accomplishes this with a new public method that’s added to any class that inherits from it:

[Click here to view code image](#)

```
class ToDictMixin:
    def to_dict(self):
        return self._traverse_dict(self.__dict__)
```

The implementation details are straightforward and rely on dynamic attribute access using `hasattr`, dynamic type inspection with `isinstance`, and accessing the instance dictionary `__dict__`:

[Click here to view code image](#)

```
def _traverse_dict(self, instance_dict):
    output = {}
    for key, value in instance_dict.items():
        output[key] = self._traverse(key, value)
    return output
```

```

def _traverse(self, key, value):
    if isinstance(value, ToDictMixin):
        return value.to_dict()
    elif isinstance(value, dict):
        return self._traverse_dict(value)
    elif isinstance(value, list):
        return [self._traverse(key, i) for i in value]
    elif hasattr(value, '__dict__'):
        return self._traverse_dict(value.__dict__)
    else:
        return value

```

Here, I define an example class that uses the mix-in to make a dictionary representation of a binary tree:

[Click here to view code image](#)

```

class BinaryTree(ToDictMixin):
    def __init__(self, value, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

```

Translating a large number of related Python objects into a dictionary becomes easy:

[Click here to view code image](#)

```

tree = BinaryTree(10,
    left=BinaryTree(7, right=BinaryTree(9)),
    right=BinaryTree(13, left=BinaryTree(11)))
print(tree.to_dict())

>>>
{'value': 10,
 'left': {'value': 7,
          'left': None,
          'right': {'value': 9, 'left': None, 'right': None}},
 'right': {'value': 13,
           'left': {'value': 11, 'left': None, 'right': None},
           'right': None}}

```

The best part about mix-ins is that you can make their generic functionality pluggable so behaviors can be overridden when required. For example, here I define a subclass of `BinaryTree` that holds a reference to its parent. This





```
'right': None,  
'parent': None}
```

By defining `BinaryTreeWithParent._traverse`, I've also enabled any class that has an attribute of type `BinaryTreeWithParent` to automatically work with the `ToDictMixin`:

[Click here to view code image](#)

```
class NamedSubTree(ToDictMixin):  
    def __init__(self, name, tree_with_parent):  
        self.name = name  
        self.tree_with_parent = tree_with_parent  
  
my_tree = NamedSubTree('foobar', root.left.right)  
print(my_tree.to_dict()) # No infinite loop  
  
>>>  
{'name': 'foobar',  
  'tree_with_parent': {'value': 9,  
                        'left': None,  
                        'right': None,  
                        'parent': 7}}
```

Mix-ins can also be composed together. For example, say I want a mix-in that provides generic JSON serialization for any class. I can do this by assuming that a class provides a `to_dict` method (which may or may not be provided by the `ToDictMixin` class):

[Click here to view code image](#)

```
import json  
  
class JsonMixin:  
    @classmethod  
    def from_json(cls, data):  
        kwargs = json.loads(data)  
        return cls(**kwargs)  
  
    def to_json(self):  
        return json.dumps(self.to_dict())
```

Note how the `JsonMixin` class defines both instance methods and class methods. Mix-ins let you add either kind of behavior to subclasses. In this example, the only requirements of a `JsonMixin` subclass are providing a

to\_dict method and taking keyword arguments for the \_\_init\_\_ method (see [Item 23: “Provide Optional Behavior with Keyword Arguments”](#) for background).

This mix-in makes it simple to create hierarchies of utility classes that can be serialized to and from JSON with little boilerplate. For example, here I have a hierarchy of data classes representing parts of a datacenter topology:

[Click here to view code image](#)

```
class DatacenterRack(ToDictMixin, JsonMixin):
    def __init__(self, switch=None, machines=None):
        self.switch = Switch(**switch)
        self.machines = [
            Machine(**kwargs) for kwargs in machines]

class Switch(ToDictMixin, JsonMixin):
    def __init__(self, ports=None, speed=None):
        self.ports = ports
        self.speed = speed

class Machine(ToDictMixin, JsonMixin):
    def __init__(self, cores=None, ram=None, disk=None):
        self.cores = cores
        self.ram = ram
        self.disk = disk
```

Serializing these classes to and from JSON is simple. Here, I verify that the data is able to be sent round-trip through serializing and deserializing:

[Click here to view code image](#)

```
serialized = """{
    "switch": {"ports": 5, "speed": 1e9},
    "machines": [
        {"cores": 8, "ram": 32e9, "disk": 5e12},
        {"cores": 4, "ram": 16e9, "disk": 1e12},
        {"cores": 2, "ram": 4e9, "disk": 500e9}
    ]
}"""

deserialized = DatacenterRack.from_json(serialized)
roundtrip = deserialized.to_json()
assert json.loads(serialized) == json.loads(roundtrip)
```

When you use mix-ins like this, it's fine if the class you apply `JsonMixin` to already inherits from `JsonMixin` higher up in the class hierarchy. The resulting class will behave the same way, thanks to the behavior of `super`.

## Things to Remember

- ✦ Avoid using multiple inheritance with instance attributes and `__init__` if mix-in classes can achieve the same outcome.
- ✦ Use pluggable behaviors at the instance level to provide per-class customization when mix-in classes may require it.
- ✦ Mix-ins can include instance methods or class methods, depending on your needs.
- ✦ Compose mix-ins to create complex functionality from simple behaviors.

## Item 42: Prefer Public Attributes Over Private Ones

In Python, there are only two types of visibility for a class's attributes: *public* and *private*:

```
class MyObject:
    def __init__(self):
        self.public_field = 5
        self.__private_field = 10

    def get_private_field(self):
        return self.__private_field
```

Public attributes can be accessed by anyone using the dot operator on the object:

```
foo = MyObject()
assert foo.public_field == 5
```

Private fields are specified by prefixing an attribute's name with a double underscore. They can be accessed directly by methods of the containing class:

[Click here to view code image](#)

```
assert foo.get_private_field() == 10
```

However, directly accessing private fields from outside the class raises an exception:

[Click here to view code image](#)

```
foo.__private_field
```

```
>>>
```

```
Traceback ...
```

```
AttributeError: 'MyObject' object has no attribute
```

```
➔ '__private_field'
```

Class methods also have access to private attributes because they are declared within the surrounding class block:

[Click here to view code image](#)

```
class MyOtherObject:
```

```
    def __init__(self):
```

```
        self.__private_field = 71
```

```
    @classmethod
```

```
    def get_private_field_of_instance(cls, instance):
```

```
        return instance.__private_field
```

```
bar = MyOtherObject()
```

```
assert MyOtherObject.get_private_field_of_instance(bar) == 71
```

As you'd expect with private fields, a subclass can't access its parent class's private fields:

[Click here to view code image](#)

```
class MyParentObject:
```

```
    def __init__(self):
```

```
        self.__private_field = 71
```

```
class MyChildObject(MyParentObject):
```

```
    def get_private_field(self):
```

```
        return self.__private_field
```

```
baz = MyChildObject()
```

```
baz.get_private_field()
```

```
>>>
```

```
Traceback ...
```

```
AttributeError: 'MyChildObject' object has no attribute  
➔ '_MyChildObject__private_field'
```

The private attribute behavior is implemented with a simple transformation of the attribute name. When the Python compiler sees private attribute access in methods like `MyChildObject.get_private_field`, it translates the `__private_field` attribute access to use the name `_MyChildObject__private_field` instead. In the example above, `__private_field` is only defined in `MyParentObject.__init__`, which means the private attribute’s real name is `_MyParentObject__private_field`. Accessing the parent’s private attribute from the child class fails simply because the transformed attribute name doesn’t exist (`_MyChildObject__private_field` instead of `_MyParentObject__private_field`).

Knowing this scheme, you can easily access the private attributes of any class—from a subclass or externally—without asking for permission:

[Click here to view code image](#)

```
assert baz._MyParentObject__private_field == 71
```

If you look in the object’s attribute dictionary, you can see that private attributes are actually stored with the names as they appear after the transformation:

[Click here to view code image](#)

```
print(baz.__dict__)  
  
>>>  
{'_MyParentObject__private_field': 71}
```

Why doesn’t the syntax for private attributes actually enforce strict visibility? The simplest answer is one often-quoted motto of Python: “We are all consenting adults here.” What this means is that we don’t need the language to prevent us from doing what we want to do. It’s our individual choice to extend functionality as we wish and to take responsibility for the consequences of such a risk. Python programmers believe that the benefits

of being open—permitting unplanned extension of classes by default—outweigh the downsides.

Beyond that, having the ability to hook language features like attribute access (see [Item 47: “Use `\_\_getattr\_\_`, `\_\_getattribute\_\_`, and `\_\_setattr\_\_` for Lazy Attributes](#)”) enables you to mess around with the internals of objects whenever you wish. If you can do that, what is the value of Python trying to prevent private attribute access otherwise?

To minimize damage from accessing internals unknowingly, Python programmers follow a naming convention defined in the style guide (see [Item 2: “Follow the PEP 8 Style Guide](#)”). Fields prefixed by a single underscore (like `_protected_field`) are *protected* by convention, meaning external users of the class should proceed with caution.

However, many programmers who are new to Python use private fields to indicate an internal API that shouldn’t be accessed by subclasses or externally:

```
class MyStringClass:
    def __init__(self, value):
        self.__value = value

    def get_value(self):
        return str(self.__value)

foo = MyStringClass(5)
assert foo.get_value() == '5'
```

This is the wrong approach. Inevitably someone—maybe even you—will want to subclass your class to add new behavior or to work around deficiencies in existing methods (e.g., the way that `MyStringClass.get_value` always returns a string). By choosing private attributes, you’re only making subclass overrides and extensions cumbersome and brittle. Your potential subclassers will still access the private fields when they absolutely need to do so:

[Click here to view code image](#)

```
class MyIntegerSubclass(MyStringClass):
    def get_value(self):
        return int(self._MyStringClass__value)
```

```
foo = MyIntegerSubclass('5')
assert foo.get_value() == 5
```

But if the class hierarchy changes beneath you, these classes will break because the private attribute references are no longer valid. Here, the `MyIntegerSubclass` class's immediate parent, `MyStringClass`, has had another parent class added, called `MyBaseClass`:

[Click here to view code image](#)

```
class MyBaseClass:
    def __init__(self, value):
        self.__value = value

    def get_value(self):
        return self.__value

class MyStringClass(MyBaseClass):
    def get_value(self):
        return str(super().get_value()) # Updated

class MyIntegerSubclass(MyStringClass):
    def get_value(self):
        return int(self._MyStringClass__value) # Not updated
```

The `__value` attribute is now assigned in the `MyBaseClass` parent class, not the `MyStringClass` parent. This causes the private variable reference `self._MyStringClass__value` to break in `MyIntegerSubclass`:

[Click here to view code image](#)

```
foo = MyIntegerSubclass(5)
foo.get_value()

>>>
Traceback ...
AttributeError: 'MyIntegerSubclass' object has no attribute
➔ '_MyStringClass__value'
```

In general, it's better to err on the side of allowing subclasses to do more by using protected attributes. Document each protected field and explain which fields are internal APIs available to subclasses and which should be left

alone entirely. This is as much advice to other programmers as it is guidance for your future self on how to extend your own code safely:

[Click here to view code image](#)

```
class MyStringClass:
    def __init__(self, value):
        # This stores the user-supplied value for the object.
        # It should be coercible to a string. Once assigned in
        # the object it should be treated as immutable.

        self._value = value
    ...
```

The only time to seriously consider using private attributes is when you're worried about naming conflicts with subclasses. This problem occurs when a child class unwittingly defines an attribute that was already defined by its parent class:

[Click here to view code image](#)

```
class ApiClass:
    def __init__(self):
        self._value = 5

    def get(self):
        return self._value

class Child(ApiClass):
    def __init__(self):
        super().__init__()
        self._value = 'hello' # Conflicts

a = Child()
print(f'{a.get()} and {a._value} should be different')

>>>
hello and hello should be different
```

This is primarily a concern with classes that are part of a public API; the subclasses are out of your control, so you can't refactor to fix the problem. Such a conflict is especially possible with attribute names that are very common (like `value`). To reduce the risk of this issue occurring, you can use



a private attribute in the parent class to ensure that there are no attribute names that overlap with child classes:

[Click here to view code image](#)

```
class ApiClass:
    def __init__(self):
        self.__value = 5      # Double underscore

    def get(self):
        return self.__value   # Double underscore

class Child(ApiClass):
    def __init__(self):
        super().__init__()
        self._value = 'hello' # OK!
a = Child()
print(f'{a.get()} and {a._value} are different')

>>>
5 and hello are different
```

## Things to Remember

- ✦ Private attributes aren't rigorously enforced by the Python compiler.
- ✦ Plan from the beginning to allow subclasses to do more with your internal APIs and attributes instead of choosing to lock them out.
- ✦ Use documentation of protected fields to guide subclasses instead of trying to force access control with private attributes.
- ✦ Only consider using private attributes to avoid naming conflicts with subclasses that are out of your control.

## Item 43: Inherit from `collections.abc` for Custom Container Types

Much of programming in Python is defining classes that contain data and describing how such objects relate to each other. Every Python class is a container of some kind, encapsulating attributes and functionality together.

Python also provides built-in container types for managing data: lists, tuples, sets, and dictionaries.

When you're designing classes for simple use cases like sequences, it's natural to want to subclass Python's built-in `list` type directly. For example, say I want to create my own custom `list` type that has additional methods for counting the frequency of its members:

[Click here to view code image](#)

```
class FrequencyList(list):
    def __init__(self, members):
        super().__init__(members)

    def frequency(self):
        counts = {}
        for item in self:
            counts[item] = counts.get(item, 0) + 1
        return counts
```

By subclassing `list`, I get all of `list`'s standard functionality and preserve the semantics familiar to all Python programmers. I can define additional methods to provide any custom behaviors that I need:

[Click here to view code image](#)

```
foo = FrequencyList(['a', 'b', 'a', 'c', 'b', 'a', 'd'])
print('Length is', len(foo))

foo.pop()
print('After pop:', repr(foo))
print('Frequency:', foo.frequency())

>>>
Length is 7
After pop: ['a', 'b', 'a', 'c', 'b', 'a']
Frequency: {'a': 3, 'b': 2, 'c': 1}
```

Now, imagine that I want to provide an object that feels like a `list` and allows indexing but isn't a `list` subclass. For example, say that I want to provide sequence semantics (like `list` or `tuple`) for a binary tree class:

[Click here to view code image](#)

```
class BinaryNode:
    def __init__(self, value, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right
```

How do you make this class act like a sequence type? Python implements its container behaviors with instance methods that have special names.

When you access a sequence item by index:

```
bar = [1, 2, 3]
bar[0]
```

it will be interpreted as:

```
bar.__getitem__(0)
```

To make the `BinaryNode` class act like a sequence, you can provide a custom implementation of `__getitem__` (often pronounced “dunder getitem” as an abbreviation for “double underscore getitem”) that traverses the object tree depth first:

[Click here to view code image](#)

```
class IndexableNode(BinaryNode):
    def _traverse(self):
        if self.left is not None:
            yield from self.left._traverse()
        yield self
        if self.right is not None:
            yield from self.right._traverse()

    def __getitem__(self, index):
        for i, item in enumerate(self._traverse()):
            if i == index:
                return item.value
        raise IndexError(f'Index {index} is out of range')
```

You can construct your binary tree as usual:

[Click here to view code image](#)

```
tree = IndexableNode(
    10,
    left=IndexableNode(
        5,
```

```

        left=IndexableNode(2),
        right=IndexableNode(
            6,
            right=IndexableNode(7))),
right=IndexableNode(
    15,
    left=IndexableNode(11)))

```

But you can also access it like a `list` in addition to being able to traverse the tree with the `left` and `right` attributes:

[Click here to view code image](#)

```

print('LRR is', tree.left.right.right.value)
print('Index 0 is', tree[0])
print('Index 1 is', tree[1])
print('11 in the tree?', 11 in tree)
print('17 in the tree?', 17 in tree)
print('Tree is', list(tree))

```

```

>>>
LRR is 7
Index 0 is 2
Index 1 is 5
11 in the tree? True
17 in the tree? False
Tree is [2, 5, 6, 7, 10, 11, 15]

```

The problem is that implementing `__getitem__` isn't enough to provide all of the sequence semantics you'd expect from a `list` instance:

[Click here to view code image](#)

```
len(tree)
```

```

>>>
Traceback ...
TypeError: object of type 'IndexableNode' has no len()

```

The `len` built-in function requires another special method, named `__len__`, that must have an implementation for a custom sequence type:

[Click here to view code image](#)

```

class SequenceNode(IndexableNode):
    def __len__(self):

```

```

        for count, _ in enumerate(self._traverse(), 1):
            pass
        return count
tree = SequenceNode(
    10,
    left=SequenceNode(
        5,
        left=SequenceNode(2),
        right=SequenceNode(
            6,
            right=SequenceNode(7))),
    right=SequenceNode(
        15,
        left=SequenceNode(11))
)
print('Tree length is', len(tree))

>>>
Tree length is 7

```

Unfortunately, this still isn't enough for the class to fully be a valid sequence. Also missing are the `count` and `index` methods that a Python programmer would expect to see on a sequence like `list` or `tuple`. It turns out that defining your own container types is much harder than it seems.

To avoid this difficulty throughout the Python universe, the built-in `collections.abc` module defines a set of abstract base classes that provide all of the typical methods for each container type. When you subclass from these abstract base classes and forget to implement required methods, the module tells you something is wrong:

[Click here to view code image](#)

```

from collections.abc import Sequence

class BadType(Sequence):
    pass

foo = BadType()

>>>
Traceback ...
TypeError: Can't instantiate abstract class BadType with
➡abstract methods __getitem__, __len__

```

When you do implement all the methods required by an abstract base class from `collections.abc`, as I did above with `SequenceNode`, it provides all of the additional methods, like `index` and `count`, for free:

[Click here to view code image](#)

```
class BetterNode(SequenceNode, Sequence):
    pass
tree = BetterNode(
    10,
    left=BetterNode(
        5,
        left=BetterNode(2),
        right=BetterNode(
            6,
            right=BetterNode(7))),
    right=BetterNode(
        15,
        left=BetterNode(11))
)

print('Index of 7 is', tree.index(7))
print('Count of 10 is', tree.count(10))

>>>
Index of 7 is 3
Count of 10 is 1
```

The benefit of using these abstract base classes is even greater for more complex container types such as `Set` and `MutableMapping`, which have a large number of special methods that need to be implemented to match Python conventions.

Beyond the `collections.abc` module, Python uses a variety of special methods for object comparisons and sorting, which may be provided by container classes and non-container classes alike (see [Item 73: “Know How to Use `heapq` for Priority Queues](#)” for an example).

## Things to Remember

- ◆ Inherit directly from Python’s container types (like `list` or `dict`) for simple use cases.

- ◆ Beware of the large number of methods required to implement custom container types correctly.
- ◆ Have your custom container types inherit from the interfaces defined in `collections.abc` to ensure that your classes match required interfaces and behaviors.

## 6. Metaclasses and Attributes

Metaclasses are often mentioned in lists of Python's features, but few understand what they accomplish in practice. The name *metaclass* vaguely implies a concept above and beyond a class. Simply put, metaclasses let you intercept Python's `class` statement and provide special behavior each time a class is defined.

Similarly mysterious and powerful are Python's built-in features for dynamically customizing attribute accesses. Along with Python's object-oriented constructs, these facilities provide wonderful tools to ease the transition from simple classes to complex ones.

However, with these powers come many pitfalls. Dynamic attributes enable you to override objects and cause unexpected side effects. Metaclasses can create extremely bizarre behaviors that are unapproachable to newcomers. It's important that you follow the *rule of least surprise* and only use these mechanisms to implement well-understood idioms.

### Item 44: Use Plain Attributes Instead of Setter and Getter Methods

Programmers coming to Python from other languages may naturally try to implement explicit getter and setter methods in their classes:

```
class OldResistor:
    def __init__(self, ohms):
        self._ohms = ohms

    def get_ohms(self):
        return self._ohms

    def set_ohms(self, ohms):
        self._ohms = ohms
```

Using these setters and getters is simple, but it's not Pythonic:

```
r0 = OldResistor(50e3)
print('Before:', r0.get_ohms())
```



```
r0.set Ohms(10e3)
print('After: ', r0.get Ohms())
```

```
>>>
```

```
Before: 50000.0
```

```
After: 10000.0
```

Such methods are especially clumsy for operations like incrementing in place:

```
r0.set Ohms(r0.get Ohms() - 4e3)
assert r0.get Ohms() == 6e3
```

These utility methods do, however, help define the interface for a class, making it easier to encapsulate functionality, validate usage, and define boundaries. Those are important goals when designing a class to ensure that you don't break callers as the class evolves over time.

In Python, however, you never need to implement explicit setter or getter methods. Instead, you should always start your implementations with simple public attributes, as I do here:

```
class Resistor:
    def __init__(self, Ohms):
        self. Ohms = Ohms
        self.voltage = 0
        self.current = 0
```

```
r1 = Resistor(50e3)
r1. Ohms = 10e3
```

These attributes make operations like incrementing in place natural and clear:

```
r1. Ohms += 5e3
```

Later, if I decide I need special behavior when an attribute is set, I can migrate to the `@property` decorator (see [Item 26: “Define Function Decorators with `functools.wraps`”](#) for background) and its corresponding setter attribute. Here, I define a new subclass of `Resistor` that lets me vary the current by assigning the voltage property. Note that in order for this code to work properly, the names of both the setter and the getter methods must match the intended property name:

[Click here to view code image](#)

```
class VoltageResistance(Resistor):
    def __init__(self, ohms):
        super().__init__(ohms)
        self._voltage = 0

    @property
    def voltage(self):
        return self._voltage

    @voltage.setter
    def voltage(self, voltage):
        self._voltage = voltage
        self.current = self._voltage / self.ohms
```

Now, assigning the voltage property will run the voltage setter method, which in turn will update the current attribute of the object to match:

[Click here to view code image](#)

```
r2 = VoltageResistance(1e3)
print(f'Before: {r2.current:.2f} amps')
r2.voltage = 10
print(f'After: {r2.current:.2f} amps')

>>>
Before: 0.00 amps
After: 0.01 amps
```

Specifying a setter on a property also enables me to perform type checking and validation on values passed to the class. Here, I define a class that ensures all resistance values are above zero ohms:

[Click here to view code image](#)

```
class BoundedResistance(Resistor):
    def __init__(self, ohms):
        super().__init__(ohms)

    @property
    def ohms(self):
        return self._ohms

    @ohms.setter
    def ohms(self, ohms):
        if ohms <= 0:
```

```
        raise ValueError(f'ohms must be > 0; got {ohms}')
    self._ohms = ohms
```

Assigning an invalid resistance to the attribute now raises an exception:

```
r3 = BoundedResistance(1e3)
r3.ohms = 0
>>>
Traceback ...
ValueError: ohms must be > 0; got 0
```

An exception is also raised if I pass an invalid value to the constructor:

[Click here to view code image](#)

```
BoundedResistance(-5)

>>>
Traceback ...
ValueError: ohms must be > 0; got -5
```

This happens because `BoundedResistance.__init__` calls `Resistor.__init__`, which assigns `self.ohms = -5`. That assignment causes the `@ohms.setter` method from `BoundedResistance` to be called, and it immediately runs the validation code before object construction has completed.

I can even use `@property` to make attributes from parent classes immutable:

[Click here to view code image](#)

```
class FixedResistance(Resistor):
    def __init__(self, ohms):
        super().__init__(ohms)

    @property
    def ohms(self):
        return self._ohms

    @ohms.setter
    def ohms(self, ohms):
        if hasattr(self, '_ohms'):
            raise AttributeError("Ohms is immutable")
        self._ohms = ohms
```

Trying to assign to the property after construction raises an exception:

```
r4 = FixedResistance(1e3)
r4.ohms = 2e3
```

```
>>>
Traceback ...
AttributeError: Ohms is immutable
```

When you use `@property` methods to implement setters and getters, be sure that the behavior you implement is not surprising. For example, don't set other attributes in getter property methods:

[Click here to view code image](#)

```
class MysteriousResistor(Resistor):
    @property
    def ohms(self):
        self.voltage = self._ohms * self.current
        return self._ohms

    @ohms.setter
    def ohms(self, ohms):
        self._ohms = ohms
```

Setting other attributes in getter property methods leads to extremely bizarre behavior:

```
r7 = MysteriousResistor(10)
r7.current = 0.01
print(f'Before: {r7.voltage:.2f}')
r7.ohms
print(f'After: {r7.voltage:.2f}')

>>>
Before: 0.00
After: 0.10
```

The best policy is to modify only related object state in `@property.setter` methods. Be sure to also avoid any other side effects that the caller may not expect beyond the object, such as importing modules dynamically, running slow helper functions, doing I/O, or making expensive database queries. Users of a class will expect its attributes to be like any other Python object: quick and easy. Use normal methods to do anything more complex or slow.

The biggest shortcoming of `@property` is that the methods for an attribute can only be shared by subclasses. Unrelated classes can't share the same implementation. However, Python also supports *descriptors* (see [Item 46: “Use Descriptors for Reusable `@property` Methods”](#)) that enable reusable property logic and many other use cases.

## Things to Remember

- ◆ Define new class interfaces using simple public attributes and avoid defining setter and getter methods.
- ◆ Use `@property` to define special behavior when attributes are accessed on your objects, if necessary.
- ◆ Follow the rule of least surprise and avoid odd side effects in your `@property` methods.
- ◆ Ensure that `@property` methods are fast; for slow or complex work—especially involving I/O or causing side effects—use normal methods instead.

## Item 45: Consider `@property` Instead of Refactoring Attributes

The built-in `@property` decorator makes it easy for simple accesses of an instance's attributes to act smarter (see [Item 44: “Use Plain Attributes Instead of Setter and Getter Methods”](#)). One advanced but common use of `@property` is transitioning what was once a simple numerical attribute into an on-the-fly calculation. This is extremely helpful because it lets you migrate all existing usage of a class to have new behaviors without requiring any of the call sites to be rewritten (which is especially important if there's calling code that you don't control). `@property` also provides an important stopgap for improving interfaces over time.

For example, say that I want to implement a leaky bucket quota using plain Python objects. Here, the `Bucket` class represents how much quota remains and the duration for which the quota will be available:

[Click here to view code image](#)

```
from datetime import datetime, timedelta

class Bucket:
    def __init__(self, period):
        self.period_delta = timedelta(seconds=period)
        self.reset_time = datetime.now()
        self.quota = 0

    def __repr__(self):
        return f'Bucket(quota={self.quota})'
```

The leaky bucket algorithm works by ensuring that, whenever the bucket is filled, the amount of quota does not carry over from one period to the next:

[Click here to view code image](#)

```
def fill(bucket, amount):
    now = datetime.now()
    if (now - bucket.reset_time) > bucket.period_delta:
        bucket.quota = 0
        bucket.reset_time = now
    bucket.quota += amount
```

Each time a quota consumer wants to do something, it must first ensure that it can deduct the amount of quota it needs to use:

[Click here to view code image](#)

```
def deduct(bucket, amount):
    now = datetime.now()
    if (now - bucket.reset_time) > bucket.period_delta:
        return False # Bucket hasn't been filled this period
    if bucket.quota - amount < 0:
        return False # Bucket was filled, but not enough

    bucket.quota -= amount
    return True # Bucket had enough, quota consumed
```

To use this class, first I fill the bucket up:

```
bucket = Bucket(60)
fill(bucket, 100)
print(bucket)
```

```
>>>
Bucket(quota=100)
```

Then, I deduct the quota that I need:

[Click here to view code image](#)

```
if deduct(bucket, 99):
    print('Had 99 quota')
else:
    print('Not enough for 99 quota')
print(bucket)
```

```
>>>
Had 99 quota
Bucket(quota=1)
```

Eventually, I'm prevented from making progress because I try to deduct more quota than is available. In this case, the bucket's quota level remains unchanged:

```
if deduct(bucket, 3):
    print('Had 3 quota')
else:
    print('Not enough for 3 quota')
print(bucket)
```

```
>>>
Not enough for 3 quota
Bucket(quota=1)
```

The problem with this implementation is that I never know what quota level the bucket started with. The quota is deducted over the course of the period until it reaches zero. At that point, deduct will always return `False` until the bucket is refilled. When that happens, it would be useful to know whether callers to deduct are being blocked because the Bucket ran out of quota or because the Bucket never had quota during this period in the first place.

To fix this, I can change the class to keep track of the `max_quota` issued in the period and the `quota_consumed` in the period:

[Click here to view code image](#)

```
class NewBucket:
    def __init__(self, period):
```

```

self.period_delta = timedelta(seconds=period)
self.reset_time = datetime.now()
self.max_quota = 0
self.quota_consumed = 0

def __repr__(self):
    return (f'NewBucket(max_quota={self.max_quota}, '
            f'quota_consumed={self.quota_consumed})')

```

To match the previous interface of the original Bucket class, I use a @property method to compute the current level of quota on-the-fly using these new attributes:

[Click here to view code image](#)

```

@property
def quota(self):
    return self.max_quota - self.quota_consumed

```

When the quota attribute is assigned, I take special action to be compatible with the current usage of the class by the fill and deduct functions:

[Click here to view code image](#)

```

@quota.setter
def quota(self, amount):
    delta = self.max_quota - amount
    if amount == 0:
        # Quota being reset for a new period
        self.quota_consumed = 0
        self.max_quota = 0
    elif delta < 0:
        # Quota being filled for the new period
        assert self.quota_consumed == 0
        self.max_quota = amount
    else:
        # Quota being consumed during the period
        assert self.max_quota >= self.quota_consumed
        self.quota_consumed += delta

```

Rerunning the demo code from above produces the same results:

[Click here to view code image](#)

```

bucket = NewBucket(60)
print('Initial', bucket)

```



```

fill(bucket, 100)
print('Filled', bucket)

if deduct(bucket, 99):
    print('Had 99 quota')
else:
    print('Not enough for 99 quota')
print('Now', bucket)

if deduct(bucket, 3):
    print('Had 3 quota')
else:
    print('Not enough for 3 quota')

print('Still', bucket)

>>>
Initial NewBucket(max_quota=0, quota_consumed=0)
Filled NewBucket(max_quota=100, quota_consumed=0)
Had 99 quota
Now NewBucket(max_quota=100, quota_consumed=99)
Not enough for 3 quota
Still NewBucket(max_quota=100, quota_consumed=99)

```

The best part is that the code using `Bucket.quota` doesn't have to change or know that the class has changed. New usage of `Bucket` can do the right thing and access `max_quota` and `quota_consumed` directly.

I especially like `@property` because it lets you make incremental progress toward a better data model over time. Reading the `Bucket` example above, you may have thought that `fill` and `deduct` should have been implemented as instance methods in the first place. Although you're probably right (see [Item 37: “Compose Classes Instead of Nesting Many Levels of Built-in Types”](#)), in practice there are many situations in which objects start with poorly defined interfaces or act as dumb data containers. This happens when code grows over time, scope increases, multiple authors contribute without anyone considering long-term hygiene, and so on.

`@property` is a tool to help you address problems you'll come across in real-world code. Don't overuse it. When you find yourself repeatedly extending `@property` methods, it's probably time to refactor your class instead of further paving over your code's poor design.

## Things to Remember

- ✦ Use `@property` to give existing instance attributes new functionality.
- ✦ Make incremental progress toward better data models by using `@property`.
- ✦ Consider refactoring a class and all call sites when you find yourself using `@property` too heavily.

## Item 46: Use Descriptors for Reusable `@property` Methods

The big problem with the `@property` built-in (see [Item 44: “Use Plain Attributes Instead of Setter and Getter Methods”](#) and [Item 45: “Consider `@property` Instead of Refactoring Attributes”](#)) is reuse. The methods it decorates can’t be reused for multiple attributes of the same class. They also can’t be reused by unrelated classes.

For example, say I want a class to validate that the grade received by a student on a homework assignment is a percentage:

[Click here to view code image](#)

```
class Homework:
    def __init__(self):
        self._grade = 0

    @property
    def grade(self):
        return self._grade

    @grade.setter
    def grade(self, value):
        if not (0 <= value <= 100):
            raise ValueError(
                'Grade must be between 0 and 100')
        self._grade = value
```

Using `@property` makes this class easy to use:

```
galileo = Homework()
galileo.grade = 95
```

Say that I also want to give the student a grade for an exam, where the exam has multiple subjects, each with a separate grade:

[Click here to view code image](#)

```
class Exam:
    def __init__(self):
        self._writing_grade = 0
        self._math_grade = 0

    @staticmethod
    def _check_grade(value):
        if not (0 <= value <= 100):
            raise ValueError(
                'Grade must be between 0 and 100')
```

This quickly gets tedious. For each section of the exam I need to add a new @property and related validation:

```
@property
def writing_grade(self):
    return self._writing_grade

@writing_grade.setter
def writing_grade(self, value):
    self._check_grade(value)
    self._writing_grade = value

@property
def math_grade(self):
    return self._math_grade

@math_grade.setter
def math_grade(self, value):
    self._check_grade(value)
    self._math_grade = value
```

Also, this approach is not general. If I want to reuse this percentage validation in other classes beyond homework and exams, I'll need to write the @property boilerplate and \_check\_grade method over and over again.

The better way to do this in Python is to use a *descriptor*. The *descriptor protocol* defines how attribute access is interpreted by the language. A descriptor class can provide \_\_get\_\_ and \_\_set\_\_ methods that let you reuse the grade validation behavior without boilerplate. For this purpose,

descriptors are also better than mix-ins (see [Item 41: “Consider Composing Functionality with Mix-in Classes”](#)) because they let you reuse the same logic for many different attributes in a single class.

Here, I define a new class called `Exam` with class attributes that are `Grade` instances. The `Grade` class implements the descriptor protocol:

[Click here to view code image](#)

```
class Grade:
    def __get__(self, instance, instance_type):
        ...

    def __set__(self, instance, value):
        ...

class Exam:
    # Class attributes
    math_grade = Grade()
    writing_grade = Grade()
    science_grade = Grade()
```

Before I explain how the `Grade` class works, it's important to understand what Python will do when such descriptor attributes are accessed on an `Exam` instance. When I assign a property:

```
exam = Exam()
exam.writing_grade = 40
```

it is interpreted as:

[Click here to view code image](#)

```
Exam.__dict__['writing_grade'].__set__(exam, 40)
```

When I retrieve a property:

```
exam.writing_grade
```

it is interpreted as:

[Click here to view code image](#)

```
Exam.__dict__['writing_grade'].__get__(exam, Exam)
```

What drives this behavior is the `__getattr__` method of object (see [Item 47: “Use `\_\_getattr\_\_`, `\_\_getattribute\_\_`, and `\_\_setattr\_\_` for Lazy Attributes](#)”). In short, when an `Exam` instance doesn't have an attribute named `writing_grade`, Python falls back to the `Exam` class's attribute instead. If this class attribute is an object that has `__get__` and `__set__` methods, Python assumes that you want to follow the descriptor protocol.

Knowing this behavior and how I used `@property` for grade validation in the `Homework` class, here's a reasonable first attempt at implementing the grade descriptor:

[Click here to view code image](#)

```
class Grade:
    def __init__(self):
        self._value = 0

    def __get__(self, instance, instance_type):
        return self._value

    def __set__(self, instance, value):
        if not (0 <= value <= 100):
            raise ValueError(
                'Grade must be between 0 and 100')
        self._value = value
```

Unfortunately, this is wrong and results in broken behavior. Accessing multiple attributes on a single `Exam` instance works as expected:

[Click here to view code image](#)

```
class Exam:
    math_grade = Grade()
    writing_grade = Grade()
    science_grade = Grade()

first_exam = Exam()
first_exam.writing_grade = 82
first_exam.science_grade = 99
print('Writing', first_exam.writing_grade)
print('Science', first_exam.science_grade)

>>>
```

Writing 82  
Science 99

But accessing these attributes on multiple Exam instances causes unexpected behavior:

[Click here to view code image](#)

```
second_exam = Exam()
second_exam.writing_grade = 75
print(f'Second {second_exam.writing_grade} is right')
print(f'First {first_exam.writing_grade} is wrong; '
      f'should be 82')
```

```
>>>
Second 75 is right
First 75 is wrong; should be 82
```

The problem is that a single Grade instance is shared across all Exam instances for the class attribute `writing_grade`. The Grade instance for this attribute is constructed once in the program lifetime, when the Exam class is first defined, not each time an Exam instance is created.

To solve this, I need the Grade class to keep track of its value for each unique Exam instance. I can do this by saving the per-instance state in a dictionary:

[Click here to view code image](#)

```
class Grade:
    def __init__(self):
        self._values = {}

    def __get__(self, instance, instance_type):
        if instance is None:
            return self
        return self._values.get(instance, 0)

    def __set__(self, instance, value):
        if not (0 <= value <= 100):
            raise ValueError(
                'Grade must be between 0 and 100')
        self._values[instance] = value
```

This implementation is simple and works well, but there's still one gotcha: It leaks memory. The `_values` dictionary holds a reference to every instance of `Exam` ever passed to `__set__` over the lifetime of the program. This causes instances to never have their reference count go to zero, preventing cleanup by the garbage collector (see [Item 81: “Use `tracemalloc` to Understand Memory Usage and Leaks](#)” for how to detect this type of problem).

To fix this, I can use Python's `weakref` built-in module. This module provides a special class called `WeakKeyDictionary` that can take the place of the simple dictionary used for `_values`. The unique behavior of `WeakKeyDictionary` is that it removes `Exam` instances from its set of items when the Python runtime knows it's holding the instance's last remaining reference in the program. Python does the bookkeeping for me and ensures that the `_values` dictionary will be empty when all `Exam` instances are no longer in use:

[Click here to view code image](#)

```
from weakref import WeakKeyDictionary

class Grade:
    def __init__(self):
        self._values = WeakKeyDictionary()

    def __get__(self, instance, instance_type):
        ...

    def __set__(self, instance, value):
        ...
```

Using this implementation of the `Grade` descriptor, everything works as expected:

[Click here to view code image](#)

```
class Exam:
    math_grade = Grade()
    writing_grade = Grade()
    science_grade = Grade()

first_exam = Exam()
first_exam.writing_grade = 82
second_exam = Exam()
```

```

second_exam.writing_grade = 75
print(f'First {first_exam.writing_grade} is right')
print(f'Second {second_exam.writing_grade} is right')
>>>
First 82 is right
Second 75 is right

```

## Things to Remember

- ◆ Reuse the behavior and validation of `@property` methods by defining your own descriptor classes.
- ◆ Use `WeakKeyDictionary` to ensure that your descriptor classes don't cause memory leaks.
- ◆ Don't get bogged down trying to understand exactly how `__getattr__` uses the descriptor protocol for getting and setting attributes.

## Item 47: Use `__getattr__`, `__getattribute__`, and `__setattr__` for Lazy Attributes

Python's object hooks make it easy to write generic code for gluing systems together. For example, say that I want to represent the records in a database as Python objects. The database has its schema set already. My code that uses objects corresponding to those records must also know what the database looks like. However, in Python, the code that connects Python objects to the database doesn't need to explicitly specify the schema of the records; it can be generic.

How is that possible? Plain instance attributes, `@property` methods, and descriptors can't do this because they all need to be defined in advance. Python makes this dynamic behavior possible with the `__getattr__` special method. If a class defines `__getattr__`, that method is called every time an attribute can't be found in an object's instance dictionary:

```

class LazyRecord:
    def __init__(self):
        self.exists = 5

```



```
def __getattr__(self, name):
    value = f'Value for {name}'
    setattr(self, name, value)
    return value
```

Here, I access the missing property `foo`. This causes Python to call the `__getattr__` method above, which mutates the instance dictionary `__dict__`:

[Click here to view code image](#)

```
data = LazyRecord()
print('Before:', data.__dict__)
print('foo: ', data.foo)
print('After: ', data.__dict__)

>>>
Before: {'exists': 5}
foo: Value for foo
After: {'exists': 5, 'foo': 'Value for foo'}
```

Here, I add logging to `LazyRecord` to show when `__getattr__` is actually called. Note how I call `super().__getattr__()` to use the superclass's implementation of `__getattr__` in order to fetch the real property value and avoid infinite recursion (see [Item 40: “Initialize Parent Classes with `super`”](#) for background):

[Click here to view code image](#)

```
class LoggingLazyRecord(LazyRecord):
    def __getattr__(self, name):
        print(f'* Called __getattr__({name!r}), '
              f'populating instance dictionary')
        result = super().__getattr__(name)
        print(f'* Returning {result!r}')
        return result

data = LoggingLazyRecord()
print('exists: ', data.exists)
print('First foo: ', data.foo)
print('Second foo: ', data.foo)

>>>
exists: 5
* Called __getattr__('foo'), populating instance dictionary
```

```
* Returning 'Value for foo'
First foo:  Value for foo
Second foo:  Value for foo
```

The `exists` attribute is present in the instance dictionary, so `__getattr__` is never called for it. The `foo` attribute is not in the instance dictionary initially, so `__getattr__` is called the first time. But the call to `__getattr__` for `foo` also does a `setattr`, which populates `foo` in the instance dictionary. This is why the second time I access `foo`, it doesn't log a call to `__getattr__`.

This behavior is especially helpful for use cases like lazily accessing schemaless data. `__getattr__` runs once to do the hard work of loading a property; all subsequent accesses retrieve the existing result.

Say that I also want transactions in this database system. The next time the user accesses a property, I want to know whether the corresponding record in the database is still valid and whether the transaction is still open. The `__getattr__` hook won't let me do this reliably because it will use the object's instance dictionary as the fast path for existing attributes.

To enable this more advanced use case, Python has another object hook called `__getattribute__`. This special method is called every time an attribute is accessed on an object, even in cases where it *does* exist in the attribute dictionary. This enables me to do things like check global transaction state on every property access. It's important to note that such an operation can incur significant overhead and negatively impact performance, but sometimes it's worth it. Here, I define `ValidatingRecord` to log each time `__getattribute__` is called:

[Click here to view code image](#)

```
class ValidatingRecord:
    def __init__(self):
        self.exists = 5

    def __getattribute__(self, name):
        print(f'* Called __getattribute__({name!r})')
        try:
            value = super().__getattribute__(name)
            print(f'* Found {name!r}, returning {value!r}')
```

```

        return value
    except AttributeError:
        value = f'Value for {name}'
        print(f'* Setting {name!r} to {value!r}')
        setattr(self, name, value)
        return value

data = ValidatingRecord()
print('exists: ', data.exists)
print('First foo: ', data.foo)
print('Second foo: ', data.foo)

>>>
* Called __getattr__('exists')
* Found 'exists', returning 5
exists: 5
* Called __getattr__('foo')
* Setting 'foo' to 'Value for foo'
First foo: Value for foo
* Called __getattr__('foo')
* Found 'foo', returning 'Value for foo'
Second foo: Value for foo

```

In the event that a dynamically accessed property shouldn't exist, I can raise an `AttributeError` to cause Python's standard missing property behavior for both `__getattr__` and `__getattribute__`:

[Click here to view code image](#)

```

class MissingPropertyRecord:
    def __getattr__(self, name):
        if name == 'bad_name':
            raise AttributeError(f'{name} is missing')
        ...

data = MissingPropertyRecord()
data.bad_name

>>>
Traceback ...
AttributeError: bad_name is missing

```

Python code implementing generic functionality often relies on the `hasattr` built-in function to determine when properties exist, and the `getattr` built-

in function to retrieve property values. These functions also look in the instance dictionary for an attribute name before calling `__getattr__`:

[Click here to view code image](#)

```
data = LoggingLazyRecord() # Implements __getattr__
print('Before: ', data.__dict__)
print('Has first foo: ', hasattr(data, 'foo'))
print('After: ', data.__dict__)
print('Has second foo: ', hasattr(data, 'foo'))

>>>
Before: {'exists': 5}
* Called __getattr__('foo'), populating instance dictionary
* Returning 'Value for foo'
Has first foo: True
After: {'exists': 5, 'foo': 'Value for foo'}
Has second foo: True
```

In the example above, `__getattr__` is called only once. In contrast, classes that implement `__getattribute__` have that method called each time `hasattr` or `getattr` is used with an instance:

[Click here to view code image](#)

```
data = ValidatingRecord() # Implements __getattribute__
print('Has first foo: ', hasattr(data, 'foo'))
print('Has second foo: ', hasattr(data, 'foo'))

>>>
* Called __getattribute__('foo')
* Setting 'foo' to 'Value for foo'
Has first foo: True

* Called __getattribute__('foo')
* Found 'foo', returning 'Value for foo'
Has second foo: True
```

Now, say that I want to lazily push data back to the database when values are assigned to my Python object. I can do this with `__setattr__`, a similar object hook that lets you intercept arbitrary attribute assignments. Unlike when retrieving an attribute with `__getattr__` and `__getattribute__`, there's no need for two separate methods. The `__setattr__` method is

always called every time an attribute is assigned on an instance (either directly or through the `setattr` built-in function):

[Click here to view code image](#)

```
class SavingRecord:
    def __setattr__(self, name, value):
        # Save some data for the record
        ...
        super().__setattr__(name, value)
```

Here, I define a logging subclass of `SavingRecord`. Its `__setattr__` method is always called on each attribute assignment:

[Click here to view code image](#)

```
class LoggingSavingRecord(SavingRecord):
    def __setattr__(self, name, value):
        print(f'* Called __setattr__({name!r}, {value!r})')
        super().__setattr__(name, value)
```

```
data = LoggingSavingRecord()
print('Before: ', data.__dict__)
data.foo = 5
print('After: ', data.__dict__)
data.foo = 7
print('Finally:', data.__dict__)
```

```
>>>
Before: {}
* Called __setattr__('foo', 5)
After: {'foo': 5}
* Called __setattr__('foo', 7)
Finally: {'foo': 7}
```

The problem with `__getattr__` and `__setattr__` is that they're called on every attribute access for an object, even when you may not want that to happen. For example, say that I want attribute accesses on my object to actually look up keys in an associated dictionary:

[Click here to view code image](#)

```
class BrokenDictionaryRecord:
    def __init__(self, data):
        self._data = {}
    def __getattr__(self, name):
```

```
print(f'* Called __getattr__({name!r})')
return self._data[name]
```

This requires accessing `self._data` from the `__getattr__` method. However, if I actually try to do that, Python will recurse until it reaches its stack limit, and then it'll die:

[Click here to view code image](#)

```
data = BrokenDictionaryRecord({'foo': 3})
data.foo
```

```
>>>
* Called __getattr__('foo')
* Called __getattr__('_data')
* Called __getattr__('_data')
* Called __getattr__('_data')
...
Traceback ...
RecursionError: maximum recursion depth exceeded while calling
➔ a Python object
```

The problem is that `__getattr__` accesses `self._data`, which causes `__getattr__` to run again, which accesses `self._data` again, and so on. The solution is to use the `super().__getattr__` method to fetch values from the instance attribute dictionary. This avoids the recursion:

[Click here to view code image](#)

```
class DictionaryRecord:
    def __init__(self, data):
        self._data = data

    def __getattr__(self, name):
        print(f'* Called __getattr__({name!r})')
        data_dict = super().__getattr__('_data')
        return data_dict[name]

data = DictionaryRecord({'foo': 3})
print('foo: ', data.foo)

>>>
* Called __getattr__('foo')
foo: 3
```

`__setattr__` methods that modify attributes on an object also need to use `super().__setattr__` accordingly.

## Things to Remember

- ✦ Use `__getattr__` and `__setattr__` to lazily load and save attributes for an object.
- ✦ Understand that `__getattr__` only gets called when accessing a missing attribute, whereas `__getattribute__` gets called every time any attribute is accessed.
- ✦ Avoid infinite recursion in `__getattribute__` and `__setattr__` by using methods from `super()` (i.e., the object class) to access instance attributes.

## Item 48: Validate Subclasses with `__init_subclass__`

One of the simplest applications of metaclasses is verifying that a class was defined correctly. When you're building a complex class hierarchy, you may want to enforce style, require overriding methods, or have strict relationships between class attributes. Metaclasses enable these use cases by providing a reliable way to run your validation code each time a new subclass is defined.

Often a class's validation code runs in the `__init__` method, when an object of the class's type is constructed at runtime (see [Item 44: “Use Plain Attributes Instead of Setter and Getter Methods”](#) for an example). Using metaclasses for validation can raise errors much earlier, such as when the module containing the class is first imported at program startup.

Before I get into how to define a metaclass for validating subclasses, it's important to understand the metaclass action for standard objects. A metaclass is defined by inheriting from `type`. In the default case, a metaclass receives the contents of associated `class` statements in its `__new__` method. Here, I can inspect and modify the class information before the type is actually constructed:

[Click here to view code image](#)

```
class Meta(type):
    def __new__(meta, name, bases, class_dict):
        print(f'* Running {meta}.__new__ for {name}')
        print('Bases:', bases)
        print(class_dict)
        return type.__new__(meta, name, bases, class_dict)

class MyClass(metaclass=Meta):
    stuff = 123

    def foo(self):
        pass

class MySubclass(MyClass):
    other = 567

    def bar(self):
        pass
```

The metaclass has access to the name of the class, the parent classes it inherits from (bases), and all the class attributes that were defined in the class's body. All classes inherit from object, so it's not explicitly listed in the tuple of base classes:

[Click here to view code image](#)

```
>>>
* Running <class '__main__.Meta'>.__new__ for MyClass
Bases: ()
{'__module__': '__main__',
 '__qualname__': 'MyClass',
 'stuff': 123,
 'foo': <function MyClass.foo at 0x105a05280>}
* Running <class '__main__.Meta'>.__new__ for MySubclass
Bases: (<class '__main__.MyClass'>,)
{'__module__': '__main__',
 '__qualname__': 'MySubclass',
 'other': 567,
 'bar': <function MySubclass.bar at 0x105a05310>}
```

I can add functionality to the Meta.\_\_new\_\_ method in order to validate all of the parameters of an associated class before it's defined. For example, say that I want to represent any type of multisided polygon. I can do this by



defining a special validating metaclass and using it in the base class of my polygon class hierarchy. Note that it's important not to apply the same validation to the base class:

[Click here to view code image](#)

```
class ValidatePolygon(type):
    def __new__(meta, name, bases, class_dict):
        # Only validate subclasses of the Polygon class
        if bases:
            if class_dict['sides'] < 3:
                raise ValueError('Polygons need 3+ sides')
        return type.__new__(meta, name, bases, class_dict)

class Polygon(metaclass=ValidatePolygon):
    sides = None # Must be specified by subclasses

    @classmethod
    def interior_angles(cls):
        return (cls.sides - 2) * 180

class Triangle(Polygon):
    sides = 3

class Rectangle(Polygon):
    sides = 4

class Nonagon(Polygon):
    sides = 9

assert Triangle.interior_angles() == 180
assert Rectangle.interior_angles() == 360
assert Nonagon.interior_angles() == 1260
```

If I try to define a polygon with fewer than three sides, the validation will cause the class statement to fail immediately after the class statement body. This means the program will not even be able to start running when I define such a class (unless it's defined in a dynamically imported module; see [Item 88: “Know How to Break Circular Dependencies”](#) for how this can happen):

```
print('Before class')

class Line(Polygon):
    print('Before sides')
```

```

    sides = 2
    print('After sides')

print('After class')

>>>
Before class
Before sides
After sides
Traceback ...
ValueError: Polygons need 3+ sides

```

This seems like quite a lot of machinery in order to get Python to accomplish such a basic task. Luckily, Python 3.6 introduced simplified syntax—the `__init_subclass__` special class method—for achieving the same behavior while avoiding metaclasses entirely. Here, I use this mechanism to provide the same level of validation as before:

[Click here to view code image](#)

```

class BetterPolygon:
    sides = None # Must be specified by subclasses

    def __init_subclass__(cls):
        super().__init_subclass__()
        if cls.sides < 3:
            raise ValueError('Polygons need 3+ sides')

        @classmethod
        def interior_angles(cls):
            return (cls.sides - 2) * 180

class Hexagon(BetterPolygon):
    sides = 6

assert Hexagon.interior_angles() == 720

```

The code is much shorter now, and the `ValidatePolygon` metaclass is gone entirely. It's also easier to follow since I can access the `sides` attribute directly on the `cls` instance in `__init_subclass__` instead of having to go into the class's dictionary with `class_dict['sides']`. If I define an invalid subclass of `BetterPolygon`, the same exception is raised:

```

print('Before class')

```

```

class Point(BetterPolygon):
    sides = 1

print('After class')

>>>
Before class
Traceback ...
ValueError: Polygons need 3+ sides

```

Another problem with the standard Python metaclass machinery is that you can only specify a single metaclass per class definition. Here, I define a second metaclass that I'd like to use for validating the fill color used for a region (not necessarily just polygons):

[Click here to view code image](#)

```

class ValidateFilled(type):
    def __new__(meta, name, bases, class_dict):
        # Only validate subclasses of the Filled class
        if bases:
            if class_dict['color'] not in ('red', 'green'):
                raise ValueError('Fill color must be supported')
            return type.__new__(meta, name, bases, class_dict)

class Filled(metaclass=ValidateFilled):
    color = None # Must be specified by subclasses

```

When I try to use the Polygon metaclass and Filled metaclass together, I get a cryptic error message:

[Click here to view code image](#)

```

class RedPentagon(Filled, Polygon):
    color = 'red'
    sides = 5

>>>
Traceback ...
TypeError: metaclass conflict: the metaclass of a derived
→class must be a (non-strict) subclass of the metaclasses
→of all its bases

```

It's possible to fix this by creating a complex hierarchy of metaclass type definitions to layer validation:

[Click here to view code image](#)

```
class ValidatePolygon(type):
    def __new__(meta, name, bases, class_dict):
        # Only validate non-root classes
        if not class_dict.get('is_root'):
            if class_dict['sides'] < 3:
                raise ValueError('Polygons need 3+ sides')
        return type.__new__(meta, name, bases, class_dict)

class Polygon(metaclass=ValidatePolygon):
    is_root = True
    sides = None # Must be specified by subclasses

class ValidateFilledPolygon(ValidatePolygon):
    def __new__(meta, name, bases, class_dict):
        # Only validate non-root classes
        if not class_dict.get('is_root'):
            if class_dict['color'] not in ('red', 'green'):
                raise ValueError('Fill color must be supported')
        return super().__new__(meta, name, bases, class_dict)

class FilledPolygon(Polygon, metaclass=ValidateFilledPolygon):
    is_root = True
    color = None # Must be specified by subclasses
```

This requires every FilledPolygon instance to be a Polygon instance:

```
class GreenPentagon(FilledPolygon):
    color = 'green'
    sides = 5

greenie = GreenPentagon()
assert isinstance(greenie, Polygon)
```

Validation works for colors:

[Click here to view code image](#)

```
class OrangePentagon(FilledPolygon):
    color = 'orange'
    sides = 5

>>>
Traceback ...
ValueError: Fill color must be supported
```

Validation also works for number of sides:

```
class RedLine(FilledPolygon):
    color = 'red'
    sides = 2

>>>
Traceback ...
ValueError: Polygons need 3+ sides
```

However, this approach ruins composability, which is often the purpose of class validation like this (similar to mix-ins; see [Item 41: “Consider Composing Functionality with Mix-in Classes”](#)). If I want to apply the color validation logic from `ValidateFilledPolygon` to another hierarchy of classes, I’ll have to duplicate all of the logic again, which reduces code reuse and increases boilerplate.

The `__init_subclass__` special class method can also be used to solve this problem. It can be defined by multiple levels of a class hierarchy as long as the super built-in function is used to call any parent or sibling `__init_subclass__` definitions (see [Item 40: “Initialize Parent Classes with `super`”](#) for a similar example). It’s even compatible with multiple inheritance. Here, I define a class to represent region fill color that can be composed with the `BetterPolygon` class from before:

[Click here to view code image](#)

```
class Filled:
    color = None # Must be specified by subclasses

    def __init_subclass__(cls):
        super().__init_subclass__()
        if cls.color not in ('red', 'green', 'blue'):
            raise ValueError('Fills need a valid color')
```

I can inherit from both classes to define a new class. Both classes call `super().__init_subclass__()`, causing their corresponding validation logic to run when the subclass is created:

```
class RedTriangle(Filled, Polygon):
    color = 'red'
    sides = 3

ruddy = RedTriangle()
```

```
assert isinstance(ruddy, Filled)
assert isinstance(ruddy, Polygon)
```

If I specify the number of sides incorrectly, I get a validation error:

```
print('Before class')

class BlueLine(Filled, Polygon):
    color = 'blue'
    sides = 2

print('After class')

>>>
Before class
Traceback ...
ValueError: Polygons need 3+ sides
```

If I specify the color incorrectly, I also get a validation error:

```
print('Before class')

class BeigeSquare(Filled, Polygon):
    color = 'beige'
    sides = 4
print('After class')

>>>
Before class
Traceback ...
ValueError: Fills need a valid color
```

You can even use `__init_subclass__` in complex cases like diamond inheritance (see [Item 40: “Initialize Parent Classes with `super`”](#)). Here, I define a basic diamond hierarchy to show this in action:

```
class Top:
    def __init_subclass__(cls):
        super().__init_subclass__()
        print(f'Top for {cls}')

class Left(Top):
    def __init_subclass__(cls):
        super().__init_subclass__()
        print(f'Left for {cls}')

class Right(Top):
```

```

def __init_subclass__(cls):
    super().__init_subclass__()
    print(f'Right for {cls}')

class Bottom(Left, Right):
    def __init_subclass__(cls):
        super().__init_subclass__()
        print(f'Bottom for {cls}')

>>>
Top for <class '__main__.Left'>
Top for <class '__main__.Right'>
Top for <class '__main__.Bottom'>
Right for <class '__main__.Bottom'>
Left for <class '__main__.Bottom'>

```

As expected, `Top.__init_subclass__` is called only a single time for each class, even though there are two paths to it for the `Bottom` class through its `Left` and `Right` parent classes.

## Things to Remember

- ✦ The `__new__` method of metaclasses is run after the `class` statement's entire body has been processed.
- ✦ Metaclasses can be used to inspect or modify a class after it's defined but before it's created, but they're often more heavyweight than what you need.
- ✦ Use `__init_subclass__` to ensure that subclasses are well formed at the time they are defined, before objects of their type are constructed.
- ✦ Be sure to call `super().__init_subclass__` from within your class's `__init_subclass__` definition to enable validation in multiple layers of classes and multiple inheritance.

## Item 49: Register Class Existence with `__init_subclass__`

Another common use of metaclasses is to automatically register types in a program. Registration is useful for doing reverse lookups, where you need to map a simple identifier back to a corresponding class.

For example, say that I want to implement my own serialized representation of a Python object using JSON. I need a way to turn an object into a JSON string. Here, I do this generically by defining a base class that records the constructor parameters and turns them into a JSON dictionary:

[Click here to view code image](#)

```
import json

class Serializable:
    def __init__(self, *args):
        self.args = args

    def serialize(self):
        return json.dumps({'args': self.args})
```

This class makes it easy to serialize simple, immutable data structures like Point2D to a string:

[Click here to view code image](#)

```
class Point2D(Serializable):
    def __init__(self, x, y):
        super().__init__(x, y)
        self.x = x
        self.y = y

    def __repr__(self):
        return f'Point2D({self.x}, {self.y})'

point = Point2D(5, 3)
print('Object: ', point)
print('Serialized:', point.serialize())
```

```
>>>
Object:  Point2D(5, 3)
Serialized: {"args": [5, 3]}
```

Now, I need to deserialize this JSON string and construct the Point2D object it represents. Here, I define another class that can deserialize the data from its Serializable parent class:

[Click here to view code image](#)



```
class Deserializable(Serializable):
    @classmethod
    def deserialize(cls, json_data):
        params = json.loads(json_data)
        return cls(*params['args'])
```

Using Deserializable makes it easy to serialize and deserialize simple, immutable objects in a generic way:

[Click here to view code image](#)

```
class BetterPoint2D(Deserializable):
    ...
before = BetterPoint2D(5, 3)
print('Before: ', before)
data = before.serialize()
print('Serialized:', data)
after = BetterPoint2D.deserialize(data)
print('After: ', after)

>>>
Before:      Point2D(5, 3)
Serialized:  {"args": [5, 3]}
After:       Point2D(5, 3)
```

The problem with this approach is that it works only if you know the intended type of the serialized data ahead of time (e.g., Point2D, BetterPoint2D). Ideally, you'd have a large number of classes serializing to JSON and one common function that could deserialize any of them back to a corresponding Python object.

To do this, I can include the serialized object's class name in the JSON data:

[Click here to view code image](#)

```
class BetterSerializable:
    def __init__(self, *args):
        self.args = args

    def serialize(self):
        return json.dumps({
            'class': self.__class__.__name__,
            'args': self.args,
        })

    def __repr__(self):
```

```
name = self.__class__.__name__
args_str = ', '.join(str(x) for x in self.args)
return f'{name}({args_str})'
```

Then, I can maintain a mapping of class names back to constructors for those objects. The general deserialize function works for any classes passed to register\_class:

[Click here to view code image](#)

```
registry = {}

def register_class(target_class):
    registry[target_class.__name__] = target_class

def deserialize(data):
    params = json.loads(data)

    name = params['class']
    target_class = registry[name]
    return target_class(*params['args'])
```

To ensure that deserialize always works properly, I must call register\_class for every class I may want to deserialize in the future:

[Click here to view code image](#)

```
class EvenBetterPoint2D(BetterSerializable):
    def __init__(self, x, y):
        super().__init__(x, y)
        self.x = x
        self.y = y

register_class(EvenBetterPoint2D)
```

Now, I can deserialize an arbitrary JSON string without having to know which class it contains:

[Click here to view code image](#)

```
before = EvenBetterPoint2D(5, 3)
print('Before: ', before)
data = before.serialize()
print('Serialized:', data)
after = deserialize(data)
print('After: ', after)
```

```
>>>
Before:      EvenBetterPoint2D(5, 3)
Serialized: {"class": "EvenBetterPoint2D", "args": [5, 3]}
After:      EvenBetterPoint2D(5, 3)
```

The problem with this approach is that it's possible to forget to call `register_class`:

[Click here to view code image](#)

```
class Point3D(BetterSerializable):
    def __init__(self, x, y, z):
        super().__init__(x, y, z)
        self.x = x
        self.y = y
        self.z = z

# Forgot to call register_class! Whoops!
```

This causes the code to break at runtime, when I finally try to deserialize an instance of a class I forgot to register:

```
point = Point3D(5, 9, -4)
data = point.serialize()
deserialize(data)
```

```
>>>
Traceback ...
KeyError: 'Point3D'
```

Even though I chose to subclass `BetterSerializable`, I don't actually get all of its features if I forget to call `register_class` after the `class` statement body. This approach is error prone and especially challenging for beginners. The same omission can happen with *class decorators* (see [Item 51: “Prefer Class Decorators Over Metaclasses for Composable Class Extensions”](#) for when those are appropriate).

What if I could somehow act on the programmer's intent to use `BetterSerializable` and ensure that `register_class` is called in all cases? Metaclasses enable this by intercepting the `class` statement when subclasses are defined (see [Item 48: “Validate Subclasses with](#)

`__init_subclass__`” for details on the machinery). Here, I use a metaclass to register the new type immediately after the class’s body:

[Click here to view code image](#)

```
class Meta(type):
    def __new__(meta, name, bases, class_dict):
        cls = type.__new__(meta, name, bases, class_dict)
        register_class(cls)
        return cls

class RegisteredSerializable(BetterSerializable,
                             metaclass=Meta):
    pass
```

When I define a subclass of `RegisteredSerializable`, I can be confident that the call to `register_class` happened and `deserialize` will always work as expected:

[Click here to view code image](#)

```
class Vector3D(RegisteredSerializable):
    def __init__(self, x, y, z):
        super().__init__(x, y, z)
        self.x, self.y, self.z = x, y, z

before = Vector3D(10, -7, 3)
print('Before: ', before)
data = before.serialize()
print('Serialized:', data)
print('After: ', deserialize(data))

>>>
Before:      Vector3D(10, -7, 3)
Serialized:  {"class": "Vector3D", "args": [10, -7, 3]}
After:      Vector3D(10, -7, 3)
```

An even better approach is to use the `__init_subclass__` special class method. This simplified syntax, introduced in Python 3.6, reduces the visual noise of applying custom logic when a class is defined. It also makes it more approachable to beginners who may be confused by the complexity of metaclass syntax:

[Click here to view code image](#)

```

class BetterRegisteredSerializable(BetterSerializable):
    def __init_subclass__(cls):
        super().__init_subclass__()
        register_class(cls)

class Vector1D(BetterRegisteredSerializable):
    def __init__(self, magnitude):
        super().__init__(magnitude)
        self.magnitude = magnitude

before = Vector1D(6)
print('Before:      ', before)
data = before.serialize()
print('Serialized: ', data)
print('After:       ', deserialize(data))

>>>
Before:      Vector1D(6)
Serialized:  {"class": "Vector1D", "args": [6]}
After:       Vector1D(6)

```

By using `__init_subclass__` (or metaclasses) for class registration, you can ensure that you'll never miss registering a class as long as the inheritance tree is right. This works well for serialization, as I've shown, and also applies to database object-relational mappings (ORMs), extensible plug-in systems, and callback hooks.

## Things to Remember

- ✦ Class registration is a helpful pattern for building modular Python programs.
- ✦ Metaclasses let you run registration code automatically each time a base class is subclassed in a program.
- ✦ Using metaclasses for class registration helps you avoid errors by ensuring that you never miss a registration call.
- ✦ Prefer `__init_subclass__` over standard metaclass machinery because it's clearer and easier for beginners to understand.

## Item 50: Annotate Class Attributes with `__set_name__`

One more useful feature enabled by metaclasses is the ability to modify or annotate properties after a class is defined but before the class is actually used. This approach is commonly used with *descriptors* (see [Item 46: “Use Descriptors for Reusable @property Methods”](#)) to give them more introspection into how they’re being used within their containing class.

For example, say that I want to define a new class that represents a row in a customer database. I’d like to have a corresponding property on the class for each column in the database table. Here, I define a descriptor class to connect attributes to column names:

[Click here to view code image](#)

```
class Field:
    def __init__(self, name):
        self.name = name
        self.internal_name = '_' + self.name

    def __get__(self, instance, instance_type):
        if instance is None:
            return self
        return getattr(instance, self.internal_name, '')

    def __set__(self, instance, value):
        setattr(instance, self.internal_name, value)
```

With the column name stored in the `Field` descriptor, I can save all of the per-instance state directly in the instance dictionary as protected fields by using the `setattr` built-in function, and later I can load state with `getattr`. At first, this seems to be much more convenient than building descriptors with the `weakref` built-in module to avoid memory leaks.

Defining the class representing a row requires supplying the database table’s column name for each class attribute:

```
class Customer:
    # Class attributes
    first_name = Field('first_name')
    last_name = Field('last_name')
    prefix = Field('prefix')
    suffix = Field('suffix')
```

Using the class is simple. Here, you can see how the `Field` descriptors modify the instance dictionary `__dict__` as expected:

[Click here to view code image](#)

```
cust = Customer()
print(f'Before: {cust.first_name!r} {cust.__dict__}')

cust.first_name = 'Euclid'
print(f'After: {cust.first_name!r} {cust.__dict__}')

>>>
Before: '' {}
After: 'Euclid' {'_first_name': 'Euclid'}
```

But the class definition seems redundant. I already declared the name of the field for the class on the left (`'field_name ='`). Why do I also have to pass a string containing the same information to the `Field` constructor (`Field('first_name')`) on the right?

[Click here to view code image](#)

```
class Customer:
    # Left side is redundant with right side
    first_name = Field('first_name')
    ...
```

The problem is that the order of operations in the `Customer` class definition is the opposite of how it reads from left to right. First, the `Field` constructor is called as `Field('first_name')`. Then, the return value of that is assigned to `Customer.first_name`. There's no way for a `Field` instance to know upfront which class attribute it will be assigned to.

To eliminate this redundancy, I can use a metaclass. Metaclasses let you hook the class statement directly and take action as soon as a class body is finished (see [Item 48: “Validate Subclasses with `\_\_init\_subclass\_\_`”](#) for details on how they work). In this case, I can use the metaclass to assign `Field.name` and `Field.internal_name` on the descriptor automatically instead of manually specifying the field name multiple times:

[Click here to view code image](#)

```

class Meta(type):
    def __new__(meta, name, bases, class_dict):
        for key, value in class_dict.items():
            if isinstance(value, Field):
                value.name = key
                value.internal_name = '_' + key
        cls = type.__new__(meta, name, bases, class_dict)
        return cls

```

Here, I define a base class that uses the metaclass. All classes representing database rows should inherit from this class to ensure that they use the metaclass:

```

class DatabaseRow(metaclass=Meta):
    pass

```

To work with the metaclass, the `Field` descriptor is largely unchanged. The only difference is that it no longer requires arguments to be passed to its constructor. Instead, its attributes are set by the `Meta.__new__` method above:

[Click here to view code image](#)

```

class Field:
    def __init__(self):
        # These will be assigned by the metaclass.
        self.name = None
        self.internal_name = None

    def __get__(self, instance, instance_type):
        if instance is None:
            return self
        return getattr(instance, self.internal_name, '')

    def __set__(self, instance, value):
        setattr(instance, self.internal_name, value)

```

By using the metaclass, the new `DatabaseRow` base class, and the new `Field` descriptor, the class definition for a database row no longer has the redundancy from before:

```

class BetterCustomer(DatabaseRow):
    first_name = Field()
    last_name = Field()

```



```
prefix = Field()
suffix = Field()
```

The behavior of the new class is identical to the behavior of the old one:

[Click here to view code image](#)

```
cust = BetterCustomer()
print(f'Before: {cust.first_name!r} {cust.__dict__}')
cust.first_name = 'Euler'
print(f'After: {cust.first_name!r} {cust.__dict__}')

>>>
Before: '' {}
After: 'Euler' {'_first_name': 'Euler'}
```

The trouble with this approach is that you can't use the `Field` class for properties unless you also inherit from `DatabaseRow`. If you somehow forget to subclass `DatabaseRow`, or if you don't want to due to other structural requirements of the class hierarchy, the code will break:

[Click here to view code image](#)

```
class BrokenCustomer:
    first_name = Field()
    last_name = Field()
    prefix = Field()
    suffix = Field()

cust = BrokenCustomer()
cust.first_name = 'Mersenne'

>>>
Traceback ...
TypeError: attribute name must be string, not 'NoneType'
```

The solution to this problem is to use the `__set_name__` special method for descriptors. This method, introduced in Python 3.6, is called on every descriptor instance when its containing class is defined. It receives as parameters the owning class that contains the descriptor instance and the attribute name to which the descriptor instance was assigned. Here, I avoid defining a metaclass entirely and move what the `Meta.__new__` method from above was doing into `__set_name__`:

[Click here to view code image](#)

```
class Field:
    def __init__(self):
        self.name = None
        self.internal_name = None

    def __set_name__(self, owner, name):
        # Called on class creation for each descriptor
        self.name = name
        self.internal_name = '_' + name

    def __get__(self, instance, instance_type):
        if instance is None:
            return self
        return getattr(instance, self.internal_name, '')

    def __set__(self, instance, value):
        setattr(instance, self.internal_name, value)
```

Now, I can get the benefits of the `Field` descriptor without having to inherit from a specific parent class or having to use a metaclass:

[Click here to view code image](#)

```
class FixedCustomer:
    first_name = Field()
    last_name = Field()
    prefix = Field()
    suffix = Field()

cust = FixedCustomer()
print(f'Before: {cust.first_name!r} {cust.__dict__}')
cust.first_name = 'Mersenne'
print(f'After: {cust.first_name!r} {cust.__dict__}')

>>>
Before: '' {}
After: 'Mersenne' {'_first_name': 'Mersenne'}
```

## Things to Remember

- ◆ Metaclasses enable you to modify a class's attributes before the class is fully defined.

- ✦ Descriptors and metaclasses make a powerful combination for declarative behavior and runtime introspection.
- ✦ Define `__set_name__` on your descriptor classes to allow them to take into account their surrounding class and its property names.
- ✦ Avoid memory leaks and the `weakref` built-in module by having descriptors store data they manipulate directly within a class's instance dictionary.

## Item 51: Prefer Class Decorators Over Metaclasses for Composable Class Extensions

Although metaclasses allow you to customize class creation in multiple ways (see [Item 48: “Validate Subclasses with `\_\_init\_subclass\_\_`”](#) and [Item 49: “Register Class Existence with `\_\_init\_subclass\_\_`”](#)), they still fall short of handling every situation that may arise.

For example, say that I want to decorate all of the methods of a class with a helper that prints arguments, return values, and exceptions raised. Here, I define the debugging decorator (see [Item 26: “Define Function Decorators with `functools.wraps`”](#) for background):

[Click here to view code image](#)

```
from functools import wraps
```

```
def trace_func(func):
    if hasattr(func, 'tracing'): # Only decorate once
        return func

    @wraps(func)
    def wrapper(*args, **kwargs):
        result = None
        try:
            result = func(*args, **kwargs)
            return result
        except Exception as e:
            result = e
            raise
```

```

finally:
    print(f'{func.__name__}({args!r}, {kwargs!r}) -> '
          f'{result!r}')

wrapper.tracing = True
return wrapper

```

I can apply this decorator to various special methods in my new dict subclass (see [Item 43: “Inherit from `collections.abc` for Custom Container Types”](#) for background):

[Click here to view code image](#)

```

class TraceDict(dict):
    @trace_func
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    @trace_func
    def __setitem__(self, *args, **kwargs):
        return super().__setitem__(*args, **kwargs)

    @trace_func
    def __getitem__(self, *args, **kwargs):
        return super().__getitem__(*args, **kwargs)

    ...

```

And I can verify that these methods are decorated by interacting with an instance of the class:

[Click here to view code image](#)

```

trace_dict = TraceDict([('hi', 1)])
trace_dict['there'] = 2
trace_dict['hi']
try:
    trace_dict['does not exist']
except KeyError:
    pass # Expected

>>>
__init__(({ 'hi': 1}, [('hi', 1)]), {}) -> None
__setitem__(({ 'hi': 1, 'there': 2}, 'there', 2), {}) -> None
__getitem__(({ 'hi': 1, 'there': 2}, 'hi'), {}) -> 1
__getitem__(({ 'hi': 1, 'there': 2}, 'does not exist'),
➡ {}) -> KeyError('does not exist')

```

The problem with this code is that I had to redefine all of the methods that I wanted to decorate with `@trace_func`. This is redundant boilerplate that's hard to read and error prone. Further, if a new method is later added to the dict superclass, it won't be decorated unless I also define it in `TraceDict`.

One way to solve this problem is to use a metaclass to automatically decorate all methods of a class. Here, I implement this behavior by wrapping each function or method in the new type with the `trace_func` decorator:

[Click here to view code image](#)

```
import types

trace_types = (
    types.MethodType,
    types.FunctionType,
    types.BuiltinFunctionType,
    types.BuiltinMethodType,
    types.MethodDescriptorType,
    types.ClassMethodDescriptorType)

class TraceMeta(type):
    def __new__(meta, name, bases, class_dict):
        klass = super().__new__(meta, name, bases, class_dict)

        for key in dir(klass):
            value = getattr(klass, key)
            if isinstance(value, trace_types):
                wrapped = trace_func(value)
                setattr(klass, key, wrapped)

        return klass
```

Now, I can declare my dict subclass by using the `TraceMeta` metaclass and verify that it works as expected:

[Click here to view code image](#)

```
class TraceDict(dict, metaclass=TraceMeta):
    pass

trace_dict = TraceDict([('hi', 1)])
trace_dict['there'] = 2
trace_dict['hi']
```

```

try:
    trace_dict['does not exist']
except KeyError:
    pass # Expected

>>>
__new__((<class '__main__.TraceDict'>, [('hi', 1)]), {}) -> {}
__getitem__({'hi': 1, 'there': 2}, 'hi'), {}) -> 1

__getitem__({'hi': 1, 'there': 2}, 'does not exist'),
➔ {}) -> KeyError('does not exist')

```

This works, and it even prints out a call to `__new__` that was missing from my earlier implementation. What happens if I try to use `TraceMeta` when a superclass already has specified a metaclass?

[Click here to view code image](#)

```

class OtherMeta(type):
    pass

class SimpleDict(dict, metaclass=OtherMeta):
    pass

class TraceDict(SimpleDict, metaclass=TraceMeta):
    pass

>>>
Traceback ...
TypeError: metaclass conflict: the metaclass of a derived
➔class must be a (non-strict) subclass of the metaclasses
➔of all its bases

```

This fails because `TraceMeta` does not inherit from `OtherMeta`. In theory, I can use metaclass inheritance to solve this problem by having `OtherMeta` inherit from `TraceMeta`:

[Click here to view code image](#)

```

class TraceMeta(type):
    ...

class OtherMeta(TraceMeta):
    pass

class SimpleDict(dict, metaclass=OtherMeta):

```

```

    pass

class TraceDict(SimpleDict, metaclass=TraceMeta):
    pass

trace_dict = TraceDict([('hi', 1)])
trace_dict['there'] = 2
trace_dict['hi']
try:
    trace_dict['does not exist']
except KeyError:
    pass # Expected

>>>
__init_subclass__(((), {})) -> None
__new__((<class '__main__.TraceDict'>, [('hi', 1)]), {}) -> {}
__getitem__(({'hi': 1, 'there': 2}, 'hi'), {}) -> 1
__getitem__(({'hi': 1, 'there': 2}, 'does not exist'),
➡{ }) -> KeyError('does not exist')

```

But this won't work if the metaclass is from a library that I can't modify, or if I want to use multiple utility metaclasses like `TraceMeta` at the same time. The metaclass approach puts too many constraints on the class that's being modified.

To solve this problem, Python supports *class decorators*. Class decorators work just like function decorators: They're applied with the `@` symbol prefixing a function before the class declaration. The function is expected to modify or re-create the class accordingly and then return it:

```

def my_class_decorator(klass):
    klass.extra_param = 'hello'
    return klass

@my_class_decorator
class MyClass:
    pass

print(MyClass)
print(MyClass.extra_param)

>>>
<class '__main__.MyClass'>
hello

```

I can implement a class decorator to apply `trace_func` to all methods and functions of a class by moving the core of the `TraceMeta.__new__` method above into a stand-alone function. This implementation is much shorter than the metaclass version:

[Click here to view code image](#)

```
def trace(klass):
    for key in dir(klass):
        value = getattr(klass, key)
        if isinstance(value, trace_types):
            wrapped = trace_func(value)
            setattr(klass, key, wrapped)
    return klass
```

I can apply this decorator to my dict subclass to get the same behavior as I get by using the metaclass approach above:

[Click here to view code image](#)

```
@trace
class TraceDict(dict):
    pass

trace_dict = TraceDict([('hi', 1)])
trace_dict['there'] = 2
trace_dict['hi']
try:
    trace_dict['does not exist']
except KeyError:
    pass # Expected

>>>
__new__((<class '__main__.TraceDict'>, [('hi', 1)]), {}) -> {}
__getitem__(({'hi': 1, 'there': 2}, 'hi'), {}) -> 1
__getitem__(({'hi': 1, 'there': 2}, 'does not exist'),
➡{}) -> KeyError('does not exist')
```

Class decorators also work when the class being decorated already has a metaclass:

[Click here to view code image](#)

```
class OtherMeta(type):
    pass
```



```

@trace
class TraceDict(dict, metaclass=OtherMeta):
    pass

trace_dict = TraceDict([('hi', 1)])
trace_dict['there'] = 2
trace_dict['hi']
try:
    trace_dict['does not exist']
except KeyError:
    pass # Expected

>>>
__new__((<class '__main__.TraceDict'>, [('hi', 1)]), {}) -> {}
__getitem__(({'hi': 1, 'there': 2}, 'hi'), {}) -> 1
__getitem__(({'hi': 1, 'there': 2}, 'does not exist'),
➔{}) -> KeyError('does not exist')

```

When you’re looking for composable ways to extend classes, class decorators are the best tool for the job. (see [Item 73: “Know How to Use `heapq` for Priority Queues](#)” for a useful class decorator called `functools.total_ordering`.)

## Things to Remember

- ✦ A class decorator is a simple function that receives a class instance as a parameter and returns either a new class or a modified version of the original class.
- ✦ Class decorators are useful when you want to modify every method or attribute of a class with minimal boilerplate.
- ✦ Metaclasses can’t be composed together easily, while many class decorators can be used to extend the same class without conflicts.

## 7. Concurrency and Parallelism

*Concurrency* enables a computer to do many different things *seemingly* at the same time. For example, on a computer with one CPU core, the operating system rapidly changes which program is running on the single processor. In doing so, it interleaves execution of the programs, providing the illusion that the programs are running simultaneously.

*Parallelism*, in contrast, involves *actually* doing many different things at the same time. A computer with multiple CPU cores can execute multiple programs simultaneously. Each CPU core runs the instructions of a separate program, allowing each program to make forward progress during the same instant.

Within a single program, concurrency is a tool that makes it easier for programmers to solve certain types of problems. Concurrent programs enable many distinct paths of execution, including separate streams of I/O, to make forward progress in a way that seems to be both simultaneous and independent.

The key difference between parallelism and concurrency is *speedup*. When two distinct paths of execution in a program make forward progress in parallel, the time it takes to do the total work is cut in half; the speed of execution is faster by a factor of two. In contrast, concurrent programs may run thousands of separate paths of execution seemingly in parallel but provide no speedup for the total work.

Python makes it easy to write concurrent programs in a variety of styles. Threads support a relatively small amount of concurrency, while coroutines enable vast numbers of concurrent functions. Python can also be used to do parallel work through system calls, subprocesses, and C extensions. But it can be very difficult to make concurrent Python code truly run in parallel. It's important to understand how to best utilize Python in these different situations.

**Item 52: Use `subprocess` to Manage Child Processes**

Python has battle-hardened libraries for running and managing child processes. This makes it a great language for gluing together other tools, such as command-line utilities. When existing shell scripts get complicated, as they often do over time, graduating them to a rewrite in Python for the sake of readability and maintainability is a natural choice.

Child processes started by Python are able to run in parallel, enabling you to use Python to consume all of the CPU cores of a machine and maximize the throughput of programs. Although Python itself may be CPU bound (see [Item 53: “Use Threads for Blocking I/O, Avoid for Parallelism”](#)), it’s easy to use Python to drive and coordinate CPU-intensive workloads.

Python has many ways to run subprocesses (e.g., `os.popen`, `os.exec*`), but the best choice for managing child processes is to use the `subprocess` built-in module. Running a child process with `subprocess` is simple. Here, I use the module’s `run` convenience function to start a process, read its output, and verify that it terminated cleanly:

[Click here to view code image](#)

```
import subprocess

result = subprocess.run(
    ['echo', 'Hello from the child!'],
    capture_output=True,
    encoding='utf-8')

result.check_returncode() # No exception means clean exit
print(result.stdout)

>>>
Hello from the child!
```

## Note

The examples in this item assume that your system has the `echo`, `sleep`, and `openssl` commands available. On Windows, this may not be the case. Please refer to the full example code for this item to see specific directions on how to run these snippets on Windows.

Child processes run independently from their parent process, the Python interpreter. If I create a subprocess using the Popen class instead of the run function, I can poll child process status periodically while Python does other work:

[Click here to view code image](#)

```
proc = subprocess.Popen(['sleep', '1'])
while proc.poll() is None:
    print('Working...')

    # Some time-consuming work here
    ...

print('Exit status', proc.poll())

>>>
Working...
Working...
Working...
Working...
Exit status 0
```

Decoupling the child process from the parent frees up the parent process to run many child processes in parallel. Here, I do this by starting all the child processes together with Popen upfront:

[Click here to view code image](#)

```
import time

start = time.time()
sleep_procs = []
for _ in range(10):
    proc = subprocess.Popen(['sleep', '1'])
    sleep_procs.append(proc)
```

Later, I wait for them to finish their I/O and terminate with the communicate method:

[Click here to view code image](#)

```
for proc in sleep_procs:
    proc.communicate()
```

```
end = time.time()
delta = end - start
print(f'Finished in {delta:.3} seconds')
```

```
>>>
Finished in 1.05 seconds
```

If these processes ran in sequence, the total delay would be 10 seconds or more rather than the ~1 second that I measured.

You can also pipe data from a Python program into a subprocess and retrieve its output. This allows you to utilize many other programs to do work in parallel. For example, say that I want to use the `openssl` command-line tool to encrypt some data. Starting the child process with command-line arguments and I/O pipes is easy:

[Click here to view code image](#)

```
import os
def run_encrypt(data):
    env = os.environ.copy()

    env['password'] = 'zf7ShyBhZ0raQDdE/FiZpm/m/8f9X+M1'
    proc = subprocess.Popen(
        ['openssl', 'enc', '-des3', '-pass', 'env:password'],
        env=env,
        stdin=subprocess.PIPE,
        stdout=subprocess.PIPE)
    proc.stdin.write(data)
    proc.stdin.flush() # Ensure that the child gets input
    return proc
```

Here, I pipe random bytes into the encryption function, but in practice this input pipe would be fed data from user input, a file handle, a network socket, and so on:

```
procs = []
for _ in range(3):
    data = os.urandom(10)
    proc = run_encrypt(data)
    procs.append(proc)
```

The child processes run in parallel and consume their input. Here, I wait for them to finish and then retrieve their final output. The output is random encrypted bytes as expected:

```

for proc in procs:
    out, _ = proc.communicate()
    print(out[-10:])

>>>
b'\x8c(\xed\x7m1\xf0F4\xe6'
b'\x0eD\x97\xe9>\x10h{\xbd\xf0'
b'g\x93)\x14U\xa9\xdc\xdd\x04\xd2'

```

It's also possible to create chains of parallel processes, just like UNIX pipelines, connecting the output of one child process to the input of another, and so on. Here's a function that starts the `openssl` command-line tool as a subprocess to generate a Whirlpool hash of the input stream:

[Click here to view code image](#)

```

def run_hash(input_stdin):
    return subprocess.Popen(
        ['openssl', 'dgst', '-whirlpool', '-binary'],
        stdin=input_stdin,
        stdout=subprocess.PIPE)

```

Now, I can kick off one set of processes to encrypt some data and another set of processes to subsequently hash their encrypted output. Note that I have to be careful with how the `stdout` instance of the upstream process is retained by the Python interpreter process that's starting this pipeline of child processes:

[Click here to view code image](#)

```

encrypt_procs = []
hash_procs = []
for _ in range(3):
    data = os.urandom(100)

    encrypt_proc = run_encrypt(data)
    encrypt_procs.append(encrypt_proc)

    hash_proc = run_hash(encrypt_proc.stdout)
    hash_procs.append(hash_proc)

# Ensure that the child consumes the input stream and
# the communicate() method doesn't inadvertently steal
# input from the child. Also lets SIGPIPE propagate to
# the upstream process if the downstream process dies.

```

```
encrypt_proc.stdout.close()
encrypt_proc.stdout = None
```

The I/O between the child processes happens automatically once they are started. All I need to do is wait for them to finish and print the final output:

```
for proc in encrypt_procs:
    proc.communicate()
    assert proc.returncode == 0

for proc in hash_procs:
    out, _ = proc.communicate()
    print(out[-10:])
    assert proc.returncode == 0
```

```
>>>
b'\xe2j\x98h\xfd\xec\xe7T\xd84'
b'\xf3.i\x01\xd74|\xf2\x94E'
b'5_n\xc3-\xe6j\xeb[i'
```

If I'm worried about the child processes never finishing or somehow blocking on input or output pipes, I can pass the `timeout` parameter to the `communicate` method. This causes an exception to be raised if the child process hasn't finished within the time period, giving me a chance to terminate the misbehaving subprocess:

[Click here to view code image](#)

```
proc = subprocess.Popen(['sleep', '10'])
try:
    proc.communicate(timeout=0.1)

except subprocess.TimeoutExpired:
    proc.terminate()
    proc.wait()

print('Exit status', proc.poll())

>>>
Exit status -15
```

## Things to Remember

- ◆ Use the `subprocess` module to run child processes and manage their input and output streams.
- ◆ Child processes run in parallel with the Python interpreter, enabling you to maximize your usage of CPU cores.
- ◆ Use the `run` convenience function for simple usage, and the `Popen` class for advanced usage like UNIX-style pipelines.
- ◆ Use the `timeout` parameter of the `communicate` method to avoid deadlocks and hanging child processes.

## Item 53: Use Threads for Blocking I/O, Avoid for Parallelism

The standard implementation of Python is called CPython. CPython runs a Python program in two steps. First, it parses and compiles the source text into *bytecode*, which is a low-level representation of the program as 8-bit instructions. (As of Python 3.6, however, it's technically *wordcode* with 16-bit instructions, but the idea is the same.) Then, CPython runs the bytecode using a stack-based interpreter. The bytecode interpreter has state that must be maintained and coherent while the Python program executes. CPython enforces coherence with a mechanism called the *global interpreter lock* (GIL).

Essentially, the GIL is a mutual-exclusion lock (mutex) that prevents CPython from being affected by preemptive multithreading, where one thread takes control of a program by interrupting another thread. Such an interruption could corrupt the interpreter state (e.g., garbage collection reference counts) if it comes at an unexpected time. The GIL prevents these interruptions and ensures that every bytecode instruction works correctly with the CPython implementation and its C-extension modules.

The GIL has an important negative side effect. With programs written in languages like C++ or Java, having multiple threads of execution means that a program could utilize multiple CPU cores at the same time. Although Python supports multiple threads of execution, the GIL causes only one of them to ever make forward progress at a time. This means that when you



reach for threads to do parallel computation and speed up your Python programs, you will be sorely disappointed.

For example, say that I want to do something computationally intensive with Python. Here, I use a naive number factorization algorithm as a proxy:

```
def factorize(number):
    for i in range(1, number + 1):
        if number % i == 0:
            yield i
```

Factoring a set of numbers in serial takes quite a long time:

[Click here to view code image](#)

```
import time

numbers = [2139079, 1214759, 1516637, 1852285]
start = time.time()

for number in numbers:
    list(factorize(number))

end = time.time()
delta = end - start
print(f'Took {delta:.3f} seconds')

>>>
Took 0.399 seconds
```

Using multiple threads to do this computation would make sense in other languages because I could take advantage of all the CPU cores of my computer. Let me try that in Python. Here, I define a Python thread for doing the same computation as before:

[Click here to view code image](#)

```
from threading import Thread

class FactorizeThread(Thread):
    def __init__(self, number):
        super().__init__()
        self.number = number

    def run(self):
        self.factors = list(factorize(self.number))
```

Then, I start a thread for each number to factorize in parallel:

[Click here to view code image](#)

```
start = time.time()

threads = []
for number in numbers:
    thread = FactorizeThread(number)
    thread.start()
    threads.append(thread)
```

Finally, I wait for all of the threads to finish:

```
for thread in threads:
    thread.join()

end = time.time()
delta = end - start
print(f'Took {delta:.3f} seconds')

>>>
Took 0.446 seconds
```

Surprisingly, this takes even longer than running `factorize` in serial. With one thread per number, you might expect less than a 4x speedup in other languages due to the overhead of creating threads and coordinating with them. You might expect only a 2x speedup on the dualcore machine I used to run this code. But you wouldn't expect the performance of these threads to be worse when there are multiple CPUs to utilize. This demonstrates the effect of the GIL (e.g., lock contention and scheduling overhead) on programs running in the standard CPython interpreter.

There are ways to get CPython to utilize multiple cores, but they don't work with the standard `Thread` class (see [Item 64: “Consider `concurrent.futures` for True Parallelism](#)”), and they can require substantial effort. Given these limitations, why does Python support threads at all? There are two good reasons.

First, multiple threads make it easy for a program to seem like it's doing multiple things at the same time. Managing the juggling act of simultaneous tasks is difficult to implement yourself (see [Item 56: “Know How to Recognize When Concurrency Is Necessary](#)” for an example). With threads,

you can leave it to Python to run your functions concurrently. This works because CPython ensures a level of fairness between Python threads of execution, even though only one of them makes forward progress at a time due to the GIL.

The second reason Python supports threads is to deal with blocking I/O, which happens when Python does certain types of system calls.

A Python program uses system calls to ask the computer's operating system to interact with the external environment on its behalf. Blocking I/O includes things like reading and writing files, interacting with networks, communicating with devices like displays, and so on. Threads help handle blocking I/O by insulating a program from the time it takes for the operating system to respond to requests.

For example, say that I want to send a signal to a remote-controlled helicopter through a serial port. I'll use a slow system call (`select`) as a proxy for this activity. This function asks the operating system to block for 0.1 seconds and then return control to my program, which is similar to what would happen when using a synchronous serial port:

[Click here to view code image](#)

```
import select
import socket

def slow_systemcall():
    select.select([socket.socket()], [], [], 0.1)
```

Running this system call in serial requires a linearly increasing amount of time:

```
start = time.time()

for _ in range(5):
    slow_systemcall()

end = time.time()
delta = end - start
print(f'Took {delta:.3f} seconds')

>>>
Took 0.510 seconds
```

The problem is that while the `slow_systemcall` function is running, my program can't make any other progress. My program's main thread of execution is blocked on the `select` system call. This situation is awful in practice. You need to be able to compute your helicopter's next move while you're sending it a signal; otherwise, it'll crash. When you find yourself needing to do blocking I/O and computation simultaneously, it's time to consider moving your system calls to threads.

Here, I run multiple invocations of the `slow_systemcall` function in separate threads. This would allow me to communicate with multiple serial ports (and helicopters) at the same time while leaving the main thread to do whatever computation is required:

[Click here to view code image](#)

```
start = time.time()
threads = []
for _ in range(5):
    thread = Thread(target=slow_systemcall)
    thread.start()
    threads.append(thread)
```

With the threads started, here I do some work to calculate the next helicopter move before waiting for the system call threads to finish:

[Click here to view code image](#)

```
def compute_helicopter_location(index):
    ...

for i in range(5):
    compute_helicopter_location(i)

for thread in threads:
    thread.join()

end = time.time()
delta = end - start
print(f'Took {delta:.3f} seconds')

>>>
Took 0.108 seconds
```

The parallel time is ~5x less than the serial time. This shows that all the system calls will run in parallel from multiple Python threads even though they're limited by the GIL. The GIL prevents my Python code from running in parallel, but it doesn't have an effect on system calls. This works because Python threads release the GIL just before they make system calls, and they reacquire the GIL as soon as the system calls are done.

There are many other ways to deal with blocking I/O besides using threads, such as the `asyncio` built-in module, and these alternatives have important benefits. But those options might require extra work in refactoring your code to fit a different model of execution (see [Item 60: “Achieve Highly Concurrent I/O with Coroutines”](#) and [Item 62: “Mix Threads and Coroutines to Ease the Transition to `asyncio`”](#)). Using threads is the simplest way to do blocking I/O in parallel with minimal changes to your program.

## Things to Remember

- ✦ Python threads can't run in parallel on multiple CPU cores because of the global interpreter lock (GIL).
- ✦ Python threads are still useful despite the GIL because they provide an easy way to do multiple things seemingly at the same time.
- ✦ Use Python threads to make multiple system calls in parallel. This allows you to do blocking I/O at the same time as computation.

## Item 54: Use `Lock` to Prevent Data Races in Threads

After learning about the global interpreter lock (GIL) (see [Item 53: “Use Threads for Blocking I/O, Avoid for Parallelism”](#)), many new Python programmers assume they can forgo using mutual-exclusion locks (also called *mutexes*) in their code altogether. If the GIL is already preventing Python threads from running on multiple CPU cores in parallel, it must also act as a lock for a program's data structures, right? Some testing on types like lists and dictionaries may even show that this assumption appears to hold.

But beware, this is not truly the case. The GIL will not protect you. Although only one Python thread runs at a time, a thread's operations on data structures can be interrupted between any two bytecode instructions in the Python interpreter. This is dangerous if you access the same objects from multiple threads simultaneously. The invariants of your data structures could be violated at practically any time because of these interruptions, leaving your program in a corrupted state.

For example, say that I want to write a program that counts many things in parallel, like sampling light levels from a whole network of sensors. If I want to determine the total number of light samples over time, I can aggregate them with a new class:

```
class Counter:
    def __init__(self):
        self.count = 0

    def increment(self, offset):
        self.count += offset
```

Imagine that each sensor has its own worker thread because reading from the sensor requires blocking I/O. After each sensor measurement, the worker thread increments the counter up to a maximum number of desired readings:

[Click here to view code image](#)

```
def worker(sensor_index, how_many, counter):
    for _ in range(how_many):
        # Read from the sensor
        ...
        counter.increment(1)
```

Here, I run one worker thread for each sensor in parallel and wait for them all to finish their readings:

[Click here to view code image](#)

```
from threading import Thread

how_many = 10**5
counter = Counter()
```

```

threads = []
for i in range(5):
    thread = Thread(target=worker,
                    args=(i, how_many, counter))
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()

expected = how_many * 5
found = counter.count
print(f'Counter should be {expected}, got {found}')

>>>
Counter should be 500000, got 246760

```

This seemed straightforward, and the outcome should have been obvious, but the result is way off! What happened here? How could something so simple go so wrong, especially since only one Python interpreter thread can run at a time?

The Python interpreter enforces fairness between all of the threads that are executing to ensure they get roughly equal processing time. To do this, Python suspends a thread as it's running and resumes another thread in turn. The problem is that you don't know exactly when Python will suspend your threads. A thread can even be paused seemingly halfway through what looks like an atomic operation. That's what happened in this case.

The body of the counter object's increment method looks simple, and is equivalent to this statement from the perspective of the worker thread:

```
counter.count += 1
```

But the += operator used on an object attribute actually instructs Python to do three separate operations behind the scenes. The statement above is equivalent to this:

```

value = getattr(counter, 'count')
result = value + 1
setattr(counter, 'count', result)

```

Python threads incrementing the counter can be suspended between any two of these operations. This is problematic if the way the operations interleave

causes old versions of value to be assigned to the counter. Here's an example of bad interaction between two threads, A and B:

```
# Running in Thread A
value_a = getattr(counter, 'count')
# Context switch to Thread B
value_b = getattr(counter, 'count')
result_b = value_b + 1
setattr(counter, 'count', result_b)
# Context switch back to Thread A
result_a = value_a + 1
setattr(counter, 'count', result_a)
```

Thread B interrupted thread A before it had completely finished. Thread B ran and finished, but then thread A resumed mid-execution, overwriting all of thread B's progress in incrementing the counter. This is exactly what happened in the light sensor example above.

To prevent data races like these, and other forms of data structure corruption, Python includes a robust set of tools in the `threading` built-in module. The simplest and most useful of them is the `Lock` class, a mutual-exclusion lock (mutex).

By using a lock, I can have the `Counter` class protect its current value against simultaneous accesses from multiple threads. Only one thread will be able to acquire the lock at a time. Here, I use a `with` statement to acquire and release the lock; this makes it easier to see which code is executing while the lock is held (see [Item 66: “Consider `contextlib` and `with` Statements for Reusable `try/finally` Behavior](#)” for background):

```
from threading import Lock

class LockingCounter:
    def __init__(self):
        self.lock = Lock()
        self.count = 0

    def increment(self, offset):
        with self.lock:
            self.count += offset
```

Now, I run the worker threads as before but use a `LockingCounter` instead:



[Click here to view code image](#)

```
counter = LockingCounter()

for i in range(5):
    thread = Thread(target=worker,
                    args=(i, how_many, counter))
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()

expected = how_many * 5
found = counter.count
print(f'Counter should be {expected}, got {found}')
```

>>>  
Counter should be 500000, got 500000

The result is exactly what I expect. Lock solved the problem.

## Things to Remember

- ✦ Even though Python has a global interpreter lock, you're still responsible for protecting against data races between the threads in your programs.
- ✦ Your programs will corrupt their data structures if you allow multiple threads to modify the same objects without mutual-exclusion locks (mutexes).
- ✦ Use the Lock class from the threading built-in module to enforce your program's invariants between multiple threads.

## Item 55: Use `queue` to Coordinate Work Between Threads

Python programs that do many things concurrently often need to coordinate their work. One of the most useful arrangements for concurrent work is a pipeline of functions.

A pipeline works like an assembly line used in manufacturing. Pipelines have many phases in serial, with a specific function for each phase. New

pieces of work are constantly being added to the beginning of the pipeline. The functions can operate concurrently, each working on the piece of work in its phase. The work moves forward as each function completes until there are no phases remaining. This approach is especially good for work that includes blocking I/O or subprocesses—activities that can easily be parallelized using Python (see [Item 53: “Use Threads for Blocking I/O, Avoid for Parallelism”](#)).

For example, say I want to build a system that will take a constant stream of images from my digital camera, resize them, and then add them to a photo gallery online. Such a program could be split into three phases of a pipeline. New images are retrieved in the first phase. The downloaded images are passed through the resize function in the second phase. The resized images are consumed by the upload function in the final phase.

Imagine that I’ve already written Python functions that execute the phases: download, resize, upload. How do I assemble a pipeline to do the work concurrently?

```
def download(item):  
    ...  
  
def resize(item):  
    ...  
  
def upload(item):  
    ...
```

The first thing I need is a way to hand off work between the pipeline phases. This can be modeled as a thread-safe producer–consumer queue (see [Item 54: “Use Lock to Prevent Data Races in Threads”](#) to understand the importance of thread safety in Python; see [Item 71: “Prefer deque for Producer–Consumer Queues”](#) to understand queue performance):

```
from collections import deque  
from threading import Lock  
  
class MyQueue:  
    def __init__(self):  
        self.items = deque()  
        self.lock = Lock()
```

The producer, my digital camera, adds new images to the end of the deque of pending items:

```
def put(self, item):  
    with self.lock:  
        self.items.append(item)
```

The consumer, the first phase of the processing pipeline, removes images from the front of the deque of pending items:

```
def get(self):  
    with self.lock:  
        return self.items.popleft()
```

Here, I represent each phase of the pipeline as a Python thread that takes work from one queue like this, runs a function on it, and puts the result on another queue. I also track how many times the worker has checked for new input and how much work it's completed:

[Click here to view code image](#)

```
from threading import Thread  
import time  
  
class Worker(Thread):  
    def __init__(self, func, in_queue, out_queue):  
        super().__init__()  
        self.func = func  
        self.in_queue = in_queue  
        self.out_queue = out_queue  
        self.polled_count = 0  
        self.work_done = 0
```

The trickiest part is that the worker thread must properly handle the case where the input queue is empty because the previous phase hasn't completed its work yet. This happens where I catch the `IndexError` exception below. You can think of this as a holdup in the assembly line:

[Click here to view code image](#)

```
def run(self):  
    while True:  
        self.polled_count += 1  
        try:  
            item = self.in_queue.get()
```

```

except IndexError:
    time.sleep(0.01) # No work to do
else:
    result = self.func(item)
    self.out_queue.put(result)
    self.work_done += 1

```

Now, I can connect the three phases together by creating the queues for their coordination points and the corresponding worker threads:

[Click here to view code image](#)

```

download_queue = MyQueue()
resize_queue = MyQueue()
upload_queue = MyQueue()
done_queue = MyQueue()
threads = [
    Worker(download, download_queue, resize_queue),
    Worker(resize, resize_queue, upload_queue),
    Worker(upload, upload_queue, done_queue),
]

```

I can start the threads and then inject a bunch of work into the first phase of the pipeline. Here, I use a plain object instance as a proxy for the real data required by the download function:

```

for thread in threads:
    thread.start()

for _ in range(1000):
    download_queue.put(object())

```

Now, I wait for all of the items to be processed by the pipeline and end up in the done\_queue:

[Click here to view code image](#)

```

while len(done_queue.items) < 1000:
    # Do something useful while waiting
    ...

```

This runs properly, but there's an interesting side effect caused by the threads polling their input queues for new work. The tricky part, where I catch `IndexError` exceptions in the `run` method, executes a large number of times:

[Click here to view code image](#)

```
processed = len(done_queue.items)
polled = sum(t.polled_count for t in threads)
print(f'Processed {processed} items after '
      f'polling {polled} times')

>>>
Processed 1000 items after polling 3035 times
```

When the worker functions vary in their respective speeds, an earlier phase can prevent progress in later phases, backing up the pipeline. This causes later phases to starve and constantly check their input queues for new work in a tight loop. The outcome is that worker threads waste CPU time doing nothing useful; they're constantly raising and catching `IndexError` exceptions.

But that's just the beginning of what's wrong with this implementation. There are three more problems that you should also avoid. First, determining that all of the input work is complete requires yet another busy wait on the `done_queue`. Second, in `worker`, the `run` method will execute forever in its busy loop. There's no obvious way to signal to a worker thread that it's time to exit.

Third, and worst of all, a backup in the pipeline can cause the program to crash arbitrarily. If the first phase makes rapid progress but the second phase makes slow progress, then the queue connecting the first phase to the second phase will constantly increase in size. The second phase won't be able to keep up. Given enough time and input data, the program will eventually run out of memory and die.

The lesson here isn't that pipelines are bad; it's that it's hard to build a good producer-consumer queue yourself. So why even try?

## queue to the Rescue

The `queue` class from the `queue` built-in module provides all of the functionality you need to solve the problems outlined above.

`queue` eliminates the busy waiting in the worker by making the `get` method block until new data is available. For example, here I start a thread that

waits for some input data on a queue:

[Click here to view code image](#)

```
from queue import Queue

my_queue = Queue()

def consumer():
    print('Consumer waiting')
    my_queue.get()          # Runs after put() below
    print('Consumer done')

thread = Thread(target=consumer)
thread.start()
```

Even though the thread is running first, it won't finish until an item is put on the queue instance and the get method has something to return:

[Click here to view code image](#)

```
print('Producer putting')
my_queue.put(object())    # Runs before get() above
print('Producer done')
thread.join()

>>>
Consumer waiting
Producer putting
Producer done
Consumer done
```

To solve the pipeline backup issue, the queue class lets you specify the maximum amount of pending work to allow between two phases.

This buffer size causes calls to put to block when the queue is already full. For example, here I define a thread that waits for a while before consuming a queue:

[Click here to view code image](#)

```
my_queue = Queue(1)      # Buffer size of 1

def consumer():
    time.sleep(0.1)       # Wait
    my_queue.get()        # Runs second
```

```

print('Consumer got 1')
my_queue.get()           # Runs fourth
print('Consumer got 2')
print('Consumer done')

```

```

thread = Thread(target=consumer)
thread.start()

```

The wait should allow the producer thread to put both objects on the queue before the consumer thread ever calls get. But the queue size is one. This means the producer adding items to the queue will have to wait for the consumer thread to call get at least once before the second call to put will stop blocking and add the second item to the queue:

[Click here to view code image](#)

```

my_queue.put(object())    # Runs first
print('Producer put 1')
my_queue.put(object())    # Runs third
print('Producer put 2')
print('Producer done')
thread.join()

```

```

>>>
Producer put 1
Consumer got 1
Producer put 2
Producer done
Consumer got 2
Consumer done

```

The queue class can also track the progress of work using the task\_done method. This lets you wait for a phase's input queue to drain and eliminates the need to poll the last phase of a pipeline (as with the done\_queue above). For example, here I define a consumer thread that calls task\_done when it finishes working on an item:

[Click here to view code image](#)

```

in_queue = Queue()
def consumer():
    print('Consumer waiting')
    work = in_queue.get()    # Runs second
    print('Consumer working')
    # Doing work

```

```
...
print('Consumer done')
in_queue.task_done()          # Runs third
```

```
thread = Thread(target=consumer)
thread.start()
```

Now, the producer code doesn't have to join the consumer thread or poll. The producer can just wait for the `in_queue` to finish by calling `join` on the queue instance. Even once it's empty, the `in_queue` won't be joinable until after `task_done` is called for every item that was ever enqueued:

[Click here to view code image](#)

```
print('Producer putting')
in_queue.put(object())      # Runs first
print('Producer waiting')
in_queue.join()             # Runs fourth
print('Producer done')
thread.join()
```

```
>>>
Consumer waiting
Producer putting
Producer waiting
Consumer working
Consumer done
Producer done
```

I can put all these behaviors together into a queue subclass that also tells the worker thread when it should stop processing. Here, I define a `close` method that adds a special *sentinel* item to the queue that indicates there will be no more input items after it:

```
class ClosableQueue(Queue):
    SENTINEL = object()

    def close(self):
        self.put(self.SENTINEL)
```

Then, I define an iterator for the queue that looks for this special object and stops iteration when it's found. This `__iter__` method also calls `task_done` at appropriate times, letting me track the progress of work on the queue (see



[Item 31: “Be Defensive When Iterating Over Arguments”](#) for details about `__iter__`):

[Click here to view code image](#)

```
def __iter__(self):
    while True:
        item = self.get()
        try:
            if item is self.SENTINEL:
                return # Cause the thread to exit
            yield item
        finally:
            self.task_done()
```

Now, I can redefine my worker thread to rely on the behavior of the `ClosableQueue` class. The thread will exit when the for loop is exhausted:

[Click here to view code image](#)

```
class StoppableWorker(Thread):
    def __init__(self, func, in_queue, out_queue):
        super().__init__()
        self.func = func
        self.in_queue = in_queue
        self.out_queue = out_queue

    def run(self):
        for item in self.in_queue:
            result = self.func(item)
            self.out_queue.put(result)
```

I re-create the set of worker threads using the new worker class:

[Click here to view code image](#)

```
download_queue = ClosableQueue()
resize_queue = ClosableQueue()
upload_queue = ClosableQueue()
done_queue = ClosableQueue()
threads = [
    StoppableWorker(download, download_queue, resize_queue),
    StoppableWorker(resize, resize_queue, upload_queue),
    StoppableWorker(upload, upload_queue, done_queue),
]
```

After running the worker threads as before, I also send the stop signal after all the input work has been injected by closing the input queue of the first phase:

```
for thread in threads:
    thread.start()

for _ in range(1000):
    download_queue.put(object())

download_queue.close()
```

Finally, I wait for the work to finish by joining the queues that connect the phases. Each time one phase is done, I signal the next phase to stop by closing its input queue. At the end, the done\_queue contains all of the output objects, as expected:

[Click here to view code image](#)

```
download_queue.join()
resize_queue.close()
resize_queue.join()
upload_queue.close()
upload_queue.join()
print(done_queue.qsize(), 'items finished')

for thread in threads:
    thread.join()

>>>
1000 items finished
```

This approach can be extended to use multiple worker threads per phase, which can increase I/O parallelism and speed up this type of program significantly. To do this, first I define some helper functions that start and stop multiple threads. The way stop\_threads works is by calling close on each input queue once per consuming thread, which ensures that all of the workers exit cleanly:

[Click here to view code image](#)

```
def start_threads(count, *args):
    threads = [StoppableWorker(*args) for _ in range(count)]
    for thread in threads:
```

```

        thread.start()
    return threads

def stop_threads(closable_queue, threads):
    for _ in threads:
        closable_queue.close()

    closable_queue.join()

    for thread in threads:
        thread.join()

```

Then, I connect the pieces together as before, putting objects to process into the top of the pipeline, joining queues and threads along the way, and finally consuming the results:

[Click here to view code image](#)

```

download_queue = ClosableQueue()
resize_queue = ClosableQueue()
upload_queue = ClosableQueue()
done_queue = ClosableQueue()

download_threads = start_threads(
    3, download, download_queue, resize_queue)
resize_threads = start_threads(
    4, resize, resize_queue, upload_queue)
upload_threads = start_threads(
    5, upload, upload_queue, done_queue)

for _ in range(1000):
    download_queue.put(object())

stop_threads(download_queue, download_threads)
stop_threads(resize_queue, resize_threads)
stop_threads(upload_queue, upload_threads)

print(done_queue.qsize(), 'items finished')

>>>
1000 items finished

```

Although queue works well in this case of a linear pipeline, there are many other situations for which there are better tools that you should consider (see [Item 60: “Achieve Highly Concurrent I/O with Coroutines”](#)).

## Things to Remember

- ◆ Pipelines are a great way to organize sequences of work—especially I/O-bound programs—that run concurrently using multiple Python threads.
- ◆ Be aware of the many problems in building concurrent pipelines: busy waiting, how to tell workers to stop, and potential memory explosion.
- ◆ The `Queue` class has all the facilities you need to build robust pipelines: blocking operations, buffer sizes, and joining.

## Item 56: Know How to Recognize When Concurrency Is Necessary

Inevitably, as the scope of a program grows, it also becomes more complicated. Dealing with expanding requirements in a way that maintains clarity, testability, and efficiency is one of the most difficult parts of programming. Perhaps the hardest type of change to handle is moving from a single-threaded program to one that needs multiple concurrent lines of execution.

Let me demonstrate how you might encounter this problem with an example. Say that I want to implement Conway's Game of Life, a classic illustration of finite state automata. The rules of the game are simple: You have a two-dimensional grid of an arbitrary size. Each cell in the grid can either be alive or empty:

```
ALIVE = '*'
EMPTY = '-'
```

The game progresses one tick of the clock at a time. Every tick, each cell counts how many of its neighboring eight cells are still alive. Based on its neighbor count, a cell decides if it will keep living, die, or regenerate. (I'll explain the specific rules further below.) Here's an example of a  $5 \times 5$  Game of Life grid after four generations with time going to the right:

[Click here to view code image](#)

0	1	2	3	4
-----	-----	-----	-----	-----
-*----	--*--	--**--	--*--	-----
--**--	--**--	-*----	-*----	-**--
---*-	--**--	--**--	--*--	-----
-----	-----	-----	-----	-----

I can represent the state of each cell with a simple container class. The class must have methods that allow me to get and set the value of any coordinate. Coordinates that are out of bounds should wrap around, making the grid act like an infinite looping space:

[Click here to view code image](#)

```
class Grid:
    def __init__(self, height, width):
        self.height = height
        self.width = width
        self.rows = []
        for _ in range(self.height):
            self.rows.append([EMPTY] * self.width)

    def get(self, y, x):
        return self.rows[y % self.height][x % self.width]

    def set(self, y, x, state):
        self.rows[y % self.height][x % self.width] = state

    def __str__(self):
        ...
```

To see this class in action, I can create a `Grid` instance and set its initial state to a classic shape called a glider:

```
grid = Grid(5, 9)
grid.set(0, 3, ALIVE)
grid.set(1, 4, ALIVE)
grid.set(2, 2, ALIVE)
grid.set(2, 3, ALIVE)
grid.set(2, 4, ALIVE)
print(grid)
```

```
>>>
---*-----
----*-----
--***-----
```

-----  
-----

Now, I need a way to retrieve the status of neighboring cells. I can do this with a helper function that queries the grid and returns the count of living neighbors. I use a simple function for the get parameter instead of passing in a whole Grid instance in order to reduce coupling (see [Item 38: “Accept Functions Instead of Classes for Simple Interfaces”](#) for more about this approach):

[Click here to view code image](#)

```
def count_neighbors(y, x, get):
    n_ = get(y - 1, x + 0) # North
    ne = get(y - 1, x + 1) # Northeast
    e_ = get(y + 0, x + 1) # East
    se = get(y + 1, x + 1) # Southeast
    s_ = get(y + 1, x + 0) # South
    sw = get(y + 1, x - 1) # Southwest
    w_ = get(y + 0, x - 1) # West
    nw = get(y - 1, x - 1) # Northwest
    neighbor_states = [n_, ne, e_, se, s_, sw, w_, nw]
    count = 0
    for state in neighbor_states:
        if state == ALIVE:
            count += 1
    return count
```

Now, I define the simple logic for Conway’s Game of Life, based on the game’s three rules: Die if a cell has fewer than two neighbors, die if a cell has more than three neighbors, or become alive if an empty cell has exactly three neighbors:

[Click here to view code image](#)

```
def game_logic(state, neighbors):
    if state == ALIVE:
        if neighbors < 2:
            return EMPTY # Die: Too few
        elif neighbors > 3:
            return EMPTY # Die: Too many
    else:
        if neighbors == 3:
            return ALIVE # Regenerate
    return state
```

I can connect `count_neighbors` and `game_logic` together in another function that transitions the state of a cell. This function will be called each generation to figure out a cell's current state, inspect the neighboring cells around it, determine what its next state should be, and update the resulting grid accordingly. Again, I use a function interface for `set` instead of passing in the `Grid` instance to make this code more decoupled:

[Click here to view code image](#)

```
def step_cell(y, x, get, set):
    state = get(y, x)
    neighbors = count_neighbors(y, x, get)
    next_state = game_logic(state, neighbors)
    set(y, x, next_state)
```

Finally, I can define a function that progresses the whole grid of cells forward by a single step and then returns a new grid containing the state for the next generation. The important detail here is that I need all dependent functions to call the `get` method on the previous generation's `Grid` instance, and to call the `set` method on the next generation's `Grid` instance. This is how I ensure that all of the cells move in lockstep, which is an essential part of how the game works. This is easy to achieve because I used function interfaces for `get` and `set` instead of passing `Grid` instances:

[Click here to view code image](#)

```
def simulate(grid):
    next_grid = Grid(grid.height, grid.width)
    for y in range(grid.height):
        for x in range(grid.width):
            step_cell(y, x, grid.get, next_grid.set)
    return next_grid
```

Now, I can progress the grid forward one generation at a time. You can see how the glider moves down and to the right on the grid based on the simple rules from the `game_logic` function:

[Click here to view code image](#)

```
class ColumnPrinter:
    ...
```

```
columns = ColumnPrinter()
for i in range(5):
    columns.append(str(grid))
    grid = simulate(grid)
```

```
print(columns)
```

```
>>>
```

0	1	2	3	4
---*-----	-----	-----	-----	-----
---*-----	--*-*-----	-----*	---*-----	-----*
--***-----	---**-----	--*-*-----	---**-----	-----*
-----	---*-----	---**-----	---**-----	-----***
-----	-----	-----	-----	-----

This works great for a program that can run in one thread on a single machine. But imagine that the program's requirements have changed—as I alluded to above—and now I need to do some I/O (e.g., with a socket) from within the `game_logic` function. For example, this might be required if I'm trying to build a massively multiplayer online game where the state transitions are determined by a combination of the grid state and communication with other players over the Internet.

How can I extend this implementation to support such functionality? The simplest thing to do is to add blocking I/O directly into the `game_logic` function:

[Click here to view code image](#)

```
def game_logic(state, neighbors):
    ...
    # Do some blocking input/output in here:
    data = my_socket.recv(100)
    ...
```

The problem with this approach is that it's going to slow down the whole program. If the latency of the I/O required is 100 milliseconds (i.e., a reasonably good cross-country, round-trip latency on the Internet), and there are 45 cells in the grid, then each generation will take a minimum of 4.5 seconds to evaluate because each cell is processed serially in the `simulate` function. That's far too slow and will make the game unplayable. It also



scales poorly: If I later wanted to expand the grid to 10,000 cells, I would need over 15 minutes to evaluate each generation.

The solution is to do the I/O in parallel so each generation takes roughly 100 milliseconds, regardless of how big the grid is. The process of spawning a concurrent line of execution for each unit of work—a cell in this case—is called *fan-out*. Waiting for all of those concurrent units of work to finish before moving on to the next phase in a coordinated process—a generation in this case—is called *fan-in*.

Python provides many built-in tools for achieving fan-out and fan-in with various trade-offs. You should understand the pros and cons of each approach and choose the best tool for the job, depending on the situation. See the items that follow for details based on this Game of Life example program (Item 57: “[Avoid Creating New Thread Instances for On-demand Fan-out](#),” Item 58: “[Understand How Using queue for Concurrency Requires Refactoring](#),” Item 59: “[Consider ThreadPoolExecutor When Threads Are Necessary for Concurrency](#),” and Item 60: “[Achieve Highly Concurrent I/O with Coroutines](#)”).

## Things to Remember

- ◆ A program often grows to require multiple concurrent lines of execution as its scope and complexity increases.
- ◆ The most common types of concurrency coordination are fan-out (generating new units of concurrency) and fan-in (waiting for existing units of concurrency to complete).
- ◆ Python has many different ways of achieving fan-out and fan-in.

## Item 57: Avoid Creating New Thread Instances for On-demand Fan-out

Threads are the natural first tool to reach for in order to do parallel I/O in Python (see Item 53: “[Use Threads for Blocking I/O, Avoid for Parallelism](#)”). However, they have significant downsides when you try to use them for fanning out to many concurrent lines of execution.

To demonstrate this, I'll continue with the Game of Life example from before (see [Item 56: “Know How to Recognize When Concurrency Is Necessary”](#) for background and the implementations of various functions and classes below). I'll use threads to solve the latency problem caused by doing I/O in the `game_logic` function. To begin, threads require coordination using locks to ensure that assumptions within data structures are maintained properly. I can create a subclass of the `Grid` class that adds locking behavior so an instance can be used by multiple threads simultaneously:

[Click here to view code image](#)

```
from threading import Lock

ALIVE = '*'
EMPTY = '-'

class Grid:
    ...

class LockingGrid(Grid):
    def __init__(self, height, width):
        super().__init__(height, width)
        self.lock = Lock()

    def __str__(self):
        with self.lock:
            return super().__str__()

    def get(self, y, x):
        with self.lock:
            return super().get(y, x)

    def set(self, y, x, state):
        with self.lock:
            return super().set(y, x, state)
```

Then, I can reimplement the `simulate` function to *fan out* by creating a thread for each call to `step_cell`. The threads will run in parallel and won't have to wait on each other's I/O. I can then *fan in* by waiting for all of the threads to complete before moving on to the next generation:

[Click here to view code image](#)

```

from threading import Thread

def count_neighbors(y, x, get):
    ...

def game_logic(state, neighbors):
    ...
    # Do some blocking input/output in here:
    data = my_socket.recv(100)
    ...

def step_cell(y, x, get, set):
    state = get(y, x)
    neighbors = count_neighbors(y, x, get)
    next_state = game_logic(state, neighbors)
    set(y, x, next_state)

def simulate_threaded(grid):
    next_grid = LockingGrid(grid.height, grid.width)

    threads = []
    for y in range(grid.height):
        for x in range(grid.width):
            args = (y, x, grid.get, next_grid.set)
            thread = Thread(target=step_cell, args=args)
            thread.start() # Fan out
            threads.append(thread)

    for thread in threads:
        thread.join() # Fan in

    return next_grid

```

I can run this code using the same implementation of `step_cell` and the same driving code as before with only two lines changed to use the `LockingGrid` and `simulate_threaded` implementations:

[Click here to view code image](#)

```

class ColumnPrinter:
    ...

grid = LockingGrid(5, 9) # Changed
grid.set(0, 3, ALIVE)
grid.set(1, 4, ALIVE)
grid.set(2, 2, ALIVE)
grid.set(2, 3, ALIVE)
grid.set(2, 4, ALIVE)

```

```

columns = ColumnPrinter()
for i in range(5):
    columns.append(str(grid))
    grid = simulate_threaded(grid) # Changed

print(columns)

```

```

>>>
      0      |      1      |      2      |      3      |      4
----*-----|-----*-----|-----*-----|-----*-----|-----*-----
----*-----|----*-*-----|-----*-----|----*-----|-----*-----
--***-----|----**-----|----*-*-----|----**-----|-----*-----
-----*-----|----*-----|----**-----|----**-----|-----***-----
-----*-----|-----*-----|-----*-----|-----*-----|-----*-----

```

This works as expected, and the I/O is now parallelized between the threads. However, this code has three big problems:

- The Thread instances require special tools to coordinate with each other safely (see [Item 54: “Use Lock to Prevent Data Races in Threads”](#)). This makes the code that uses threads harder to reason about than the procedural, single-threaded code from before. This complexity makes threaded code more difficult to extend and maintain over time.
- Threads require a lot of memory—about 8 MB per executing thread. On many computers, that amount of memory doesn’t matter for the 45 threads I’d need in this example. But if the game grid had to grow to 10,000 cells, I would need to create that many threads, which couldn’t even fit in the memory of my machine. Running a thread per concurrent activity just won’t work.
- Starting a thread is costly, and threads have a negative performance impact when they run due to context switching between them. In this case, all of the threads are started and stopped each generation of the game, which has high overhead and will increase latency beyond the expected I/O time of 100 milliseconds.

This code would also be very difficult to debug if something went wrong. For example, imagine that the `game_logic` function raises an exception,

which is highly likely due to the generally flaky nature of I/O:

[Click here to view code image](#)

```
def game_logic(state, neighbors):  
    ...  
    raise OSError('Problem with I/O')  
    ...
```

I can test what this would do by running a Thread instance pointed at this function and redirecting the `sys.stderr` output from the program to an in-memory StringIO buffer:

[Click here to view code image](#)

```
import contextlib  
import io  
  
fake_stderr = io.StringIO()  
with contextlib.redirect_stderr(fake_stderr):  
    thread = Thread(target=game_logic, args=(ALIVE, 3))  
    thread.start()  
    thread.join()  
  
print(fake_stderr.getvalue())
```

```
>>>  
Exception in thread Thread-226:  
Traceback (most recent call last):  
  File "threading.py", line 917, in _bootstrap_inner  
    self.run()  
  File "threading.py", line 865, in run  
    self._target(*self._args, **self._kwargs)  
  File "example.py", line 193, in game_logic  
    raise OSError('Problem with I/O')  
OSError: Problem with I/O
```

An `OSError` exception is raised as expected, but somehow the code that created the Thread and called `join` on it is unaffected. How can this be? The reason is that the Thread class will independently catch any exceptions that are raised by the target function and then write their traceback to `sys.stderr`. Such exceptions are never re-raised to the caller that started the thread in the first place.

Given all of these issues, it's clear that threads are not the solution if you need to constantly create and finish new concurrent functions. Python provides other solutions that are a better fit (see [Item 58: “Understand How Using `queue` for Concurrency Requires Refactoring](#),” [Item 59: “Consider `ThreadPoolExecutor` When Threads Are Necessary for Concurrency](#)”, and [Item 60: “Achieve Highly Concurrent I/O with Coroutines](#)”).

## Things to Remember

- ◆ Threads have many downsides: They're costly to start and run if you need a lot of them, they each require a significant amount of memory, and they require special tools like `Lock` instances for coordination.
- ◆ Threads do not provide a built-in way to raise exceptions back in the code that started a thread or that is waiting for one to finish, which makes them difficult to debug.

## Item 58: Understand How Using `queue` for Concurrency Requires Refactoring

In the previous item (see [Item 57: “Avoid Creating New Thread Instances for On-demand Fan-out](#)”) I covered the downsides of using `Thread` to solve the parallel I/O problem in the Game of Life example from earlier (see [Item 56: “Know How to Recognize When Concurrency Is Necessary](#)” for background and the implementations of various functions and classes below).

The next approach to try is to implement a threaded pipeline using the `Queue` class from the `queue` built-in module (see [Item 55: “Use `Queue` to Coordinate Work Between Threads](#)” for background; I rely on the implementations of `ClosableQueue` and `StoppableWorker` from that item in the example code below).

Here's the general approach: Instead of creating one thread per cell per generation of the Game of Life, I can create a fixed number of worker threads upfront and have them do parallelized I/O as needed. This will keep

my resource usage under control and eliminate the overhead of frequently starting new threads.

To do this, I need two `ClosableQueue` instances to use for communicating to and from the worker threads that execute the `game_logic` function:

```
from queue import Queue

class ClosableQueue(Queue):
    ...

in_queue = ClosableQueue()
out_queue = ClosableQueue()
```

I can start multiple threads that will consume items from the `in_queue`, process them by calling `game_logic`, and put the results on `out_queue`. These threads will run concurrently, allowing for parallel I/O and reduced latency for each generation:

[Click here to view code image](#)

```
from threading import Thread

class StoppableWorker(Thread):
    ...

def game_logic(state, neighbors):
    ...
    # Do some blocking input/output in here:
    data = my_socket.recv(100)
    ...

def game_logic_thread(item):
    y, x, state, neighbors = item
    try:
        next_state = game_logic(state, neighbors)
    except Exception as e:
        next_state = e
    return (y, x, next_state)

# Start the threads upfront
threads = []
for _ in range(5):
    thread = StoppableWorker(
        game_logic_thread, in_queue, out_queue)
```

```
thread.start()
threads.append(thread)
```

Now, I can redefine the `simulate` function to interact with these queues to request state transition decisions and receive corresponding responses. Adding items to `in_queue` causes *fan-out*, and consuming items from `out_queue` until it's empty causes *fan-in*:

[Click here to view code image](#)

```
ALIVE = '*'
EMPTY = '-'
```

```
class SimulationError(Exception):
    pass
```

```
class Grid:
    ...
```

```
def count_neighbors(y, x, get):
    ...
```

```
def simulate_pipeline(grid, in_queue, out_queue):
    for y in range(grid.height):
        for x in range(grid.width):
            state = grid.get(y, x)
            neighbors = count_neighbors(y, x, grid.get)
            in_queue.put((y, x, state, neighbors)) # Fan out
```

```
    in_queue.join()
    out_queue.close()
```

```
    next_grid = Grid(grid.height, grid.width)
    for item in out_queue: # Fan in
        y, x, next_state = item
        if isinstance(next_state, Exception):
            raise SimulationError(y, x) from next_state
        next_grid.set(y, x, next_state)
```

```
    return next_grid
```

The calls to `Grid.get` and `Grid.set` both happen within this new `simulate_pipeline` function, which means I can use the single-threaded implementation of `Grid` instead of the implementation that requires `Lock` instances for synchronization.



This code is also easier to debug than the Thread approach used in the previous item. If an exception occurs while doing I/O in the `game_logic` function, it will be caught, propagated to the `out_queue`, and then re-raised in the main thread:

[Click here to view code image](#)

```
def game_logic(state, neighbors):  
    ...  
    raise OSError('Problem with I/O in game_logic')  
    ...
```

```
simulate_pipeline(Grid(1, 1), in_queue, out_queue)
```

```
>>>  
Traceback ...  
OSError: Problem with I/O in game_logic
```

The above exception was the direct cause of the following  
➡exception:

```
Traceback ...  
SimulationError: (0, 0)
```

I can drive this multithreaded pipeline for repeated generations by calling `simulate_pipeline` in a loop:

[Click here to view code image](#)

```
class ColumnPrinter:  
    ...  
  
grid = Grid(5, 9)  
grid.set(0, 3, ALIVE)  
grid.set(1, 4, ALIVE)  
grid.set(2, 2, ALIVE)  
grid.set(2, 3, ALIVE)  
grid.set(2, 4, ALIVE)  
  
columns = ColumnPrinter()  
for i in range(5):  
    columns.append(str(grid))  
    grid = simulate_pipeline(grid, in_queue, out_queue)  
  
print(columns)
```

```

for thread in threads:
    in_queue.close()
for thread in threads:
    thread.join()

```

```
>>>
```

0	1	2	3	4
---*-----	-----	-----	-----	-----
---*-----	-----	---*_*-----	-----	-----*
---***-----	-----	---**-----	-----	---*_*-----
-----	-----	---*-----	-----	---**-----
-----	-----	-----	-----	-----

The results are the same as before. Although I've addressed the memory explosion problem, startup costs, and debugging issues of using threads on their own, many issues remain:

- The `simulate_pipeline` function is even harder to follow than the `simulate_threaded` approach from the previous item.
- Extra support classes were required for `ClosableQueue` and `StoppableWorker` in order to make the code easier to read, at the expense of increased complexity.
- I have to specify the amount of potential parallelism—the number of threads running `game_logic_thread`—upfront based on my expectations of the workload instead of having the system automatically scale up parallelism as needed.
- In order to enable debugging, I have to manually catch exceptions in worker threads, propagate them on a queue, and then re-raise them in the main thread.

However, the biggest problem with this code is apparent if the requirements change again. Imagine that later I needed to do I/O within the `count_neighbors` function in addition to the I/O that was needed within `game_logic`:

[Click here to view code image](#)

```

def count_neighbors(y, x, get):
    ...

```

```
# Do some blocking input/output in here:
data = my_socket.recv(100)
...
```

In order to make this parallelizable, I need to add another stage to the pipeline that runs `count_neighbors` in a thread. I need to make sure that exceptions propagate correctly between the worker threads and the main thread. And I need to use a `Lock` for the `Grid` class in order to ensure safe synchronization between the worker threads (see [Item 54: “Use Lock to Prevent Data Races in Threads”](#) for background and [Item 57: “Avoid Creating New Thread Instances for On-demand Fan-out”](#) for the implementation of `LockingGrid`):

[Click here to view code image](#)

```
def count_neighbors_thread(item):
    y, x, state, get = item
    try:
        neighbors = count_neighbors(y, x, get)
    except Exception as e:
        neighbors = e
    return (y, x, state, neighbors)

def game_logic_thread(item):
    y, x, state, neighbors = item
    if isinstance(neighbors, Exception):
        next_state = neighbors
    else:
        try:
            next_state = game_logic(state, neighbors)
        except Exception as e:
            next_state = e
    return (y, x, next_state)

class LockingGrid(Grid):
    ...
```

I have to create another set of `Queue` instances for the `count_neighbors_thread` workers and the corresponding `Thread` instances:

[Click here to view code image](#)

```
in_queue = ClosableQueue()
logic_queue = ClosableQueue()
out_queue = ClosableQueue()
```

```

threads = []

for _ in range(5):
    thread = StoppableWorker(
        count_neighbors_thread, in_queue, logic_queue)
    thread.start()
    threads.append(thread)

for _ in range(5):
    thread = StoppableWorker(
        game_logic_thread, logic_queue, out_queue)
    thread.start()
    threads.append(thread)

```

Finally, I need to update `simulate_pipeline` to coordinate the multiple phases in the pipeline and ensure that work fans out and back in correctly:

[Click here to view code image](#)

```

def simulate_phased_pipeline(
    grid, in_queue, logic_queue, out_queue):
    for y in range(grid.height):
        for x in range(grid.width):
            state = grid.get(y, x)
            item = (y, x, state, grid.get)
            in_queue.put(item)                # Fan out

    in_queue.join()
    logic_queue.join()                      # Pipeline sequencing
    out_queue.close()

    next_grid = LockingGrid(grid.height, grid.width)
    for item in out_queue:                  # Fan in
        y, x, next_state = item
        if isinstance(next_state, Exception):
            raise SimulationError(y, x) from next_state
        next_grid.set(y, x, next_state)

    return next_grid

```

With these updated implementations, now I can run the multiphase pipeline end-to-end:

[Click here to view code image](#)

```

grid = LockingGrid(5, 9)
grid.set(0, 3, ALIVE)
grid.set(1, 4, ALIVE)
grid.set(2, 2, ALIVE)
grid.set(2, 3, ALIVE)
grid.set(2, 4, ALIVE)

columns = ColumnPrinter()
for i in range(5):
    columns.append(str(grid))
    grid = simulate_phased_pipeline(
        grid, in_queue, logic_queue, out_queue)

print(columns)

for thread in threads:
    in_queue.close()
for thread in threads:
    logic_queue.close()
for thread in threads:
    thread.join()

```

```

>>>
      0      |      1      |      2      |      3      |      4
----*-----|----*-----|----*-----|----*-----|----*-----
----*-----|----*-----|----*-----|----*-----|----*-----
---***-----|---**-----|---**-----|---**-----|---**-----
-----*-----|-----*-----|-----**-----|-----**-----|-----***-----
-----*-----|-----*-----|-----**-----|-----**-----|-----***-----
-----*-----|-----*-----|-----**-----|-----**-----|-----***-----

```

Again, this works as expected, but it required a lot of changes and boilerplate. The point here is that queue does make it possible to solve fan-out and fan-in problems, but the overhead is very high. Although using queue is a better approach than using Thread instances on their own, it's still not nearly as good as some of the other tools provided by Python (see [Item 59: “Consider ThreadPoolExecutor When Threads Are Necessary for Concurrency”](#) and [Item 60: “Achieve Highly Concurrent I/O with Coroutines”](#)).

## Things to Remember

- ◆ Using queue instances with a fixed number of worker threads improves the scalability of fan-out and fan-in using threads.

- ✦ It takes a significant amount of work to refactor existing code to use Queue, especially when multiple stages of a pipeline are required.
- ✦ Using Queue fundamentally limits the total amount of I/O parallelism a program can leverage compared to alternative approaches provided by other built-in Python features and modules.

## Item 59: Consider `ThreadPoolExecutor` When Threads Are Necessary for Concurrency

Python includes the `concurrent.futures` built-in module, which provides the `ThreadPoolExecutor` class. It combines the best of the `Thread` (see [Item 57: “Avoid Creating New Thread Instances for On-demand Fan-out”](#)) and `Queue` (see [Item 58: “Understand How Using `queue` for Concurrency Requires Refactoring”](#)) approaches to solving the parallel I/O problem from the Game of Life example (see [Item 56: “Know How to Recognize When Concurrency Is Necessary”](#) for background and the implementations of various functions and classes below):

[Click here to view code image](#)

```
ALIVE = '*'
EMPTY = '-'

class Grid:
    ...

class LockingGrid(Grid):
    ...

def count_neighbors(y, x, get):
    ...

def game_logic(state, neighbors):
    ...
    # Do some blocking input/output in here:
    data = my_socket.recv(100)
    ...

def step_cell(y, x, get, set):
    state = get(y, x)
    neighbors = count_neighbors(y, x, get)
```

```
next_state = game_logic(state, neighbors)
set(y, x, next_state)
```

Instead of starting a new Thread instance for each Grid square, I can *fan out* by submitting a function to an executor that will be run in a separate thread. Later, I can wait for the result of all tasks in order to fan in:

[Click here to view code image](#)

```
from concurrent.futures import ThreadPoolExecutor

def simulate_pool(pool, grid):
    next_grid = LockingGrid(grid.height, grid.width)

    futures = []
    for y in range(grid.height):
        for x in range(grid.width):
            args = (y, x, grid.get, next_grid.set)
            future = pool.submit(step_cell, *args) # Fan out
            futures.append(future)

    for future in futures:
        future.result() # Fan in

    return next_grid
```

The threads used for the executor can be allocated in advance, which means I don't have to pay the startup cost on each execution of `simulate_pool`. I can also specify the maximum number of threads to use for the pool—using the `max_workers` parameter—to prevent the memory blow-up issues associated with the naive Thread solution to the parallel I/O problem:

[Click here to view code image](#)

```
class ColumnPrinter:
    ...

grid = LockingGrid(5, 9)
grid.set(0, 3, ALIVE)
grid.set(1, 4, ALIVE)
grid.set(2, 2, ALIVE)
grid.set(2, 3, ALIVE)
grid.set(2, 4, ALIVE)

columns = ColumnPrinter()
```

```
with ThreadPoolExecutor(max_workers=10) as pool:
    for i in range(5):
        columns.append(str(grid))
        grid = simulate_pool(pool, grid)

print(columns)
```

```
>>>
  0      |      1      |      2      |      3      |      4
---*-----|-----*-----|-----*-----|-----*-----|-----*-----
---*-----|---*-*-----|-----*-----|---*-----|-----*-----
---***-----|---**-----|---*-*-----|---**-----|-----*-----
-----|---*-----|---**-----|---**-----|---***-----
-----|-----|-----|-----|-----
```

The best part about the `ThreadPoolExecutor` class is that it automatically propagates exceptions back to the caller when the `result` method is called on the `Future` instance returned by the `submit` method:

[Click here to view code image](#)

```
def game_logic(state, neighbors):
    ...
    raise OSError('Problem with I/O')
    ...

with ThreadPoolExecutor(max_workers=10) as pool:
    task = pool.submit(game_logic, ALIVE, 3)
    task.result()

>>>
Traceback ...
OSError: Problem with I/O
```

If I needed to provide I/O parallelism for the `count_neighbors` function in addition to `game_logic`, no modifications to the program would be required since `ThreadPoolExecutor` already runs these functions concurrently as part of `step_cell`. It's even possible to achieve CPU parallelism by using the same interface if necessary (see [Item 64: “Consider `concurrent.futures` for True Parallelism](#)”).

However, the big problem that remains is the limited amount of I/O parallelism that `ThreadPoolExecutor` provides. Even if I use a `max_workers` parameter of 100, this solution still won't scale if I need 10,000+ cells in



the grid that require simultaneous I/O. `ThreadPoolExecutor` is a good choice for situations where there is no asynchronous solution (e.g., file I/O), but there are better ways to maximize I/O parallelism in many cases (see [Item 60: “Achieve Highly Concurrent I/O with Coroutines”](#)).

## Things to Remember

- ✦ `ThreadPoolExecutor` enables simple I/O parallelism with limited refactoring, easily avoiding the cost of thread startup each time fanout concurrency is required.
- ✦ Although `ThreadPoolExecutor` eliminates the potential memory blow-up issues of using threads directly, it also limits I/O parallelism by requiring `max_workers` to be specified upfront.

## Item 60: Achieve Highly Concurrent I/O with Coroutines

The previous items have tried to solve the parallel I/O problem for the Game of Life example with varying degrees of success. (see [Item 56: “Know How to Recognize When Concurrency Is Necessary”](#) for background and the implementations of various functions and classes below.) All of the other approaches fall short in their ability to handle thousands of simultaneously concurrent functions (see [Item 57: “Avoid Creating New Thread Instances for On-demand Fan-out,”](#) [Item 58: “Understand How Using `queue` for Concurrency Requires Refactoring,”](#) and [Item 59: “Consider `ThreadPoolExecutor` When Threads Are Necessary for Concurrency”](#)).

Python addresses the need for highly concurrent I/O with *coroutines*. Coroutines let you have a very large number of seemingly simultaneous functions in your Python programs. They’re implemented using the `async` and `await` keywords along with the same infrastructure that powers generators (see [Item 30: “Consider Generators Instead of Returning Lists,”](#) [Item 34: “Avoid Injecting Data into Generators with `send`,”](#) and [Item 35: “Avoid Causing State Transitions in Generators with `throw`”](#)).

The cost of starting a coroutine is a function call. Once a coroutine is active, it uses less than 1 KB of memory until it's exhausted. Like threads, coroutines are independent functions that can consume inputs from their environment and produce resulting outputs. The difference is that coroutines pause at each `await` expression and resume executing an `async` function after the pending *awaitable* is resolved (similar to how `yield` behaves in generators).

Many separate `async` functions advanced in lockstep all seem to run simultaneously, mimicking the concurrent behavior of Python threads. However, coroutines do this without the memory overhead, startup and context switching costs, or complex locking and synchronization code that's required for threads. The magical mechanism powering coroutines is the *event loop*, which can do highly concurrent I/O efficiently, while rapidly interleaving execution between appropriately written functions.

I can use coroutines to implement the Game of Life. My goal is to allow for I/O to occur within the `game_logic` function while overcoming the problems from the `Thread` and `Queue` approaches in the previous items. To do this, first I indicate that the `game_logic` function is a coroutine by defining it using `async def` instead of `def`. This will allow me to use the `await` syntax for I/O, such as an asynchronous read from a socket:

[Click here to view code image](#)

```
ALIVE = '*'
EMPTY = '-'

class Grid:
    ...

def count_neighbors(y, x, get):
    ...

async def game_logic(state, neighbors):
    ...
    # Do some input/output in here:
    data = await my_socket.read(50)
    ...
```

Similarly, I can turn `step_cell` into a coroutine by adding `async` to its definition and using `await` for the call to the `game_logic` function:

[Click here to view code image](#)

```
async def step_cell(y, x, get, set):
    state = get(y, x)
    neighbors = count_neighbors(y, x, get)
    next_state = await game_logic(state, neighbors)
    set(y, x, next_state)
```

The `simulate` function also needs to become a coroutine:

[Click here to view code image](#)

```
import asyncio

async def simulate(grid):
    next_grid = Grid(grid.height, grid.width)

    tasks = []
    for y in range(grid.height):
        for x in range(grid.width):
            task = step_cell(
                y, x, grid.get, next_grid.set)
            tasks.append(task)

    await asyncio.gather(*tasks)

    return next_grid
```

The coroutine version of the `simulate` function requires some explanation:

- Calling `step_cell` doesn't immediately run that function. Instead, it returns a coroutine instance that can be used with an `await` expression at a later time. This is similar to how generator functions that use `yield` return a generator instance when they're called instead of executing immediately. Deferring execution like this is the mechanism that causes *fan-out*.
- The `gather` function from the `asyncio` built-in library causes *fan-in*. The `await` expression on `gather` instructs the event loop to run the

step\_cell coroutines concurrently and resume execution of the simulate coroutine when all of them have been completed.

- No locks are required for the Grid instance since all execution occurs within a single thread. The I/O becomes parallelized as part of the event loop that's provided by asyncio.

Finally, I can drive this code with a one-line change to the original example. This relies on the asyncio.run function to execute the simulate coroutine in an event loop and carry out its dependent I/O:

[Click here to view code image](#)

```
class ColumnPrinter:
    ...

grid = Grid(5, 9)
grid.set(0, 3, ALIVE)
grid.set(1, 4, ALIVE)
grid.set(2, 2, ALIVE)
grid.set(2, 3, ALIVE)
grid.set(2, 4, ALIVE)

columns = ColumnPrinter()
for i in range(5):
    columns.append(str(grid))
    grid = asyncio.run(simulate(grid))    # Run the event loop

print(columns)

>>>
      0      |      1      |      2      |      3      |      4
  ---*----- | ---*----- | ---*----- | ---*----- | ---*-----
  ---*----- | --*-*----- | ---*----- | ---*----- | ---*-----
  ---***----- | ---**----- | --*-*----- | ---**----- | ---*-----
  ---*----- | ---*----- | ---**----- | ---**----- | ---***-----
  ---*----- | ---*----- | ---**----- | ---**----- | ---***-----
```

The result is the same as before. All of the overhead associated with threads has been eliminated. Whereas the Queue and ThreadPoolExecutor approaches are limited in their exception handling—merely re-raising exceptions across thread boundaries—with coroutines I can actually use the

interactive debugger to step through the code line by line (see [Item 80: “Consider Interactive Debugging with pdb”](#)):

[Click here to view code image](#)

```
async def game_logic(state, neighbors):
    ...
    raise OSError('Problem with I/O')
    ...
```

```
asyncio.run(game_logic(ALIVE, 3))
```

```
>>>
```

```
Traceback ...
```

```
OSError: Problem with I/O
```

Later, if my requirements change and I also need to do I/O from within `count_neighbors`, I can easily accomplish this by adding `async` and `await` keywords to the existing functions and call sites instead of having to restructure everything as I would have had to do if I were using `Thread` or `Queue` instances (see [Item 61: “Know How to Port Threaded I/O to asyncio”](#) for another example):

[Click here to view code image](#)

```
async def count_neighbors(y, x, get):
    ...
```

```
async def step_cell(y, x, get, set):
    state = get(y, x)
    neighbors = await count_neighbors(y, x, get)
    next_state = await game_logic(state, neighbors)
    set(y, x, next_state)
```

```
grid = Grid(5, 9)
grid.set(0, 3, ALIVE)
grid.set(1, 4, ALIVE)
grid.set(2, 2, ALIVE)
grid.set(2, 3, ALIVE)
grid.set(2, 4, ALIVE)
```

```
columns = ColumnPrinter()
for i in range(5):
    columns.append(str(grid))
    grid = asyncio.run(simulate(grid))
```

```
print(columns)
```

```
>>>
```

0	1	2	3	4
---*-----	-----	-----	-----	-----
---*-----	--*_*-----	-----*-----	---*-----	-----*-----
--***-----	---**-----	--*_*-----	-----**-----	-----*-----
-----	---*-----	---**-----	---**-----	---***-----
-----	-----	-----	-----	-----

The beauty of coroutines is that they decouple your code’s instructions for the external environment (i.e., I/O) from the implementation that carries out your wishes (i.e., the event loop). They let you focus on the logic of what you’re trying to do instead of wasting time trying to figure out how you’re going to accomplish your goals concurrently.

## Things to Remember

- ✦ Functions that are defined using the `async` keyword are called coroutines. A caller can receive the result of a dependent coroutine by using the `await` keyword.
- ✦ Coroutines provide an efficient way to run tens of thousands of functions seemingly at the same time.
- ✦ Coroutines can use fan-out and fan-in in order to parallelize I/O, while also overcoming all of the problems associated with doing I/O in threads.

## Item 61: Know How to Port Threaded I/O to `asyncio`

Once you understand the advantage of coroutines (see [Item 60: “Achieve Highly Concurrent I/O with Coroutines”](#)), it may seem daunting to port an existing codebase to use them. Luckily, Python’s support for asynchronous execution is well integrated into the language. This makes it straightforward to move code that does threaded, blocking I/O over to coroutines and asynchronous I/O.

For example, say that I have a TCP-based server for playing a game involving guessing a number. The server takes lower and upper parameters that determine the range of numbers to consider. Then, the server returns guesses for integer values in that range as they are requested by the client. Finally, the server collects reports from the client on whether each of those numbers was closer (warmer) or further away (colder) from the client's secret number.

The most common way to build this type of client/server system is by using blocking I/O and threads (see [Item 53: “Use Threads for Blocking I/O, Avoid for Parallelism”](#)). To do this, I need a helper class that can manage sending and receiving of messages. For my purposes, each line sent or received represents a command to be processed:

[Click here to view code image](#)

```
class EOFError(Exception):
    pass

class ConnectionBase:
    def __init__(self, connection):
        self.connection = connection
        self.file = connection.makefile('rb')

    def send(self, command):
        line = command + '\n'
        data = line.encode()
        self.connection.send(data)

    def receive(self):
        line = self.file.readline()
        if not line:
            raise EOFError('Connection closed')
        return line[:-1].decode()
```

The server is implemented as a class that handles one connection at a time and maintains the client's session state:

[Click here to view code image](#)

```
import random

WARMER = 'Warmer'
COLDER = 'Colder'
```

```

UNSURE = 'Unsure'
CORRECT = 'Correct'

class UnknownCommandError(Exception):
    pass

class Session(ConnectionBase):
    def __init__(self, *args):
        super().__init__(*args)
        self._clear_state(None, None)

    def _clear_state(self, lower, upper):
        self.lower = lower
        self.upper = upper
        self.secret = None
        self.guesses = []

```

It has one primary method that handles incoming commands from the client and dispatches them to methods as needed. Note that here I'm using an assignment expression (introduced in Python 3.8; see [Item 10: “Prevent Repetition with Assignment Expressions”](#)) to keep the code short:

[Click here to view code image](#)

```

def loop(self):
    while command := self.receive():
        parts = command.split(' ')
        if parts[0] == 'PARAMS':
            self.set_params(parts)
        elif parts[0] == 'NUMBER':
            self.send_number()
        elif parts[0] == 'REPORT':
            self.receive_report(parts)
        else:
            raise UnknownCommandError(command)

```

The first command sets the lower and upper bounds for the numbers that the server is trying to guess:

[Click here to view code image](#)

```

def set_params(self, parts):
    assert len(parts) == 3
    lower = int(parts[1])
    upper = int(parts[2])
    self._clear_state(lower, upper)

```



The second command makes a new guess based on the previous state that's stored in the client's session instance. Specifically, this code ensures that the server will never try to guess the same number more than once per parameter assignment:

[Click here to view code image](#)

```
def next_guess(self):
    if self.secret is not None:
        return self.secret

    while True:
        guess = random.randint(self.lower, self.upper)
        if guess not in self.guesses:
            return guess

def send_number(self):
    guess = self.next_guess()
    self.guesses.append(guess)
    self.send(format(guess))
```

The third command receives the decision from the client of whether the guess was warmer or colder, and it updates the session state accordingly:

[Click here to view code image](#)

```
def receive_report(self, parts):
    assert len(parts) == 2
    decision = parts[1]

    last = self.guesses[-1]
    if decision == CORRECT:
        self.secret = last

    print(f'Server: {last} is {decision}')
```

The client is also implemented using a stateful class:

```
import contextlib
import math

class Client(ConnectionBase):
    def __init__(self, *args):
        super().__init__(*args)
        self._clear_state()
```

```
def _clear_state(self):
    self.secret = None
    self.last_distance = None
```

The parameters of each guessing game are set using a `with` statement to ensure that state is correctly managed on the server side (see [Item 66: “Consider `contextlib` and `with` Statements for Reusable `try/finally` Behavior](#)” for background and [Item 63: “Avoid Blocking the `asyncio` Event Loop to Maximize Responsiveness](#)” for another example). This method sends the first command to the server:

[Click here to view code image](#)

```
@contextlib.contextmanager
def session(self, lower, upper, secret):
    print(f'Guess a number between {lower} and {upper}!'
          f' Shhhhhh, it\'s {secret}.')
    self.secret = secret
    self.send(f'PARAMS {lower} {upper}')
    try:
        yield
    finally:
        self._clear_state()
        self.send('PARAMS 0 -1')
```

New guesses are requested from the server, using another method that implements the second command:

[Click here to view code image](#)

```
def request_numbers(self, count):
    for _ in range(count):
        self.send('NUMBER')
        data = self.receive()
        yield int(data)
    if self.last_distance == 0:
        return
```

Whether each guess from the server was warmer or colder than the last is reported using the third command in the final method:

[Click here to view code image](#)

```
def report_outcome(self, number):
    new_distance = math.fabs(number - self.secret)
```

```

decision = UNSURE

if new_distance == 0:
    decision = CORRECT
elif self.last_distance is None:
    pass
elif new_distance < self.last_distance:
    decision = WARMER
elif new_distance > self.last_distance:
    decision = COLDER

self.last_distance = new_distance

self.send(f'REPORT {decision}')
return decision

```

I can run the server by having one thread listen on a socket and spawn additional threads to handle the new connections:

[Click here to view code image](#)

```

import socket
from threading import Thread

def handle_connection(connection):
    with connection:
        session = Session(connection)
        try:
            session.loop()
        except EOFError:
            pass

def run_server(address):
    with socket.socket() as listener:
        listener.bind(address)
        listener.listen()
        while True:
            connection, _ = listener.accept()
            thread = Thread(target=handle_connection,
                           args=(connection,),
                           daemon=True)
            thread.start()

```

The client runs in the main thread and returns the results of the guessing game to the caller. This code explicitly exercises a variety of Python language features (for loops, with statements, generators, comprehensions)

so that below I can show what it takes to port these over to using coroutines:

[Click here to view code image](#)

```
def run_client(address):
    with socket.create_connection(address) as connection:
        client = Client(connection)

        with client.session(1, 5, 3):
            results = [(x, client.report_outcome(x))
                        for x in client.request_numbers(5)]

        with client.session(10, 15, 12):
            for number in client.request_numbers(5):
                outcome = client.report_outcome(number)
                results.append((number, outcome))

    return results
```

Finally, I can glue all of this together and confirm that it works as expected:

[Click here to view code image](#)

```
def main():
    address = ('127.0.0.1', 1234)
    server_thread = Thread(
        target=run_server, args=(address,), daemon=True)
    server_thread.start()

    results = run_client(address)
    for number, outcome in results:
        print(f'Client: {number} is {outcome}')

main()
```

```
>>>
```

```
Guess a number between 1 and 5! Shhhhh, it's 3.
Server: 4 is Unsure
Server: 1 is Colder
Server: 5 is Unsure
Server: 3 is Correct
Guess a number between 10 and 15! Shhhhh, it's 12.
Server: 11 is Unsure
Server: 10 is Colder
Server: 12 is Correct
Client: 4 is Unsure
```

```
Client: 1 is Colder
Client: 5 is Unsure
Client: 3 is Correct
Client: 11 is Unsure
Client: 10 is Colder
Client: 12 is Correct
```

How much effort is needed to convert this example to using `async`, `await`, and the `asyncio` built-in module?

First, I need to update my `ConnectionBase` class to provide coroutines for `send` and `receive` instead of blocking I/O methods. I've marked each line that's changed with a `# Changed` comment to make it clear what the delta is between this new example and the code above:

[Click here to view code image](#)

```
class AsyncConnectionBase:
    def __init__(self, reader, writer):           # Changed
        self.reader = reader                    # Changed
        self.writer = writer                    # Changed

    async def send(self, command):
        line = command + '\n'
        data = line.encode()
        self.writer.write(data)                 # Changed
        await self.writer.drain()               # Changed

    async def receive(self):
        line = await self.reader.readline()     # Changed
        if not line:
            raise EOFError('Connection closed')
        return line[:-1].decode()
```

I can create another stateful class to represent the session state for a single connection. The only changes here are the class's name and inheriting from `AsyncConnectionBase` instead of `ConnectionBase`:

[Click here to view code image](#)

```
class AsyncSession(AsyncConnectionBase):       # Changed
    def __init__(self, *args):
        ...
```

```
def _clear_values(self, lower, upper):  
    ...
```

The primary entry point for the server's command processing loop requires only minimal changes to become a coroutine:

[Click here to view code image](#)

```
async def loop(self):  
    while command := await self.receive():  
        parts = command.split(' ')  
        if parts[0] == 'PARAMS':  
            self.set_params(parts)  
        elif parts[0] == 'NUMBER':  
            await self.send_number() # Changed  
        elif parts[0] == 'REPORT':  
            self.receive_report(parts)  
        else:  
            raise UnknownCommandError(command)
```

No changes are required for handling the first command:

```
def set_params(self, parts):  
    ...
```

The only change required for the second command is allowing asynchronous I/O to be used when guesses are transmitted to the client:

[Click here to view code image](#)

```
def next_guess(self):  
    ...  
  
async def send_number(self): # Changed  
    guess = self.next_guess()  
    self.guesses.append(guess)  
    await self.send(format(guess)) # Changed
```

No changes are required for processing the third command:

```
def receive_report(self, parts):  
    ...
```

Similarly, the client class needs to be reimplemented to inherit from `AsyncConnectionBase`:

[Click here to view code image](#)

```
class AsyncClient(AsyncConnectionBase):           # Changed
    def __init__(self, *args):
        ...

    def _clear_state(self):
        ...
```

The first command method for the client requires a few `async` and `await` keywords to be added. It also needs to use the `asynccontextmanager` helper function from the `contextlib` built-in module:

[Click here to view code image](#)

```
@contextlib.asynccontextmanager                 # Changed
async def session(self, lower, upper, secret):  # Changed
    print(f'Guess a number between {lower} and {upper}!'
          f' Shhhhhh, it\'s {secret}.')
    self.secret = secret
    await self.send(f'PARAMS {lower} {upper}')  # Changed
    try:
        yield
    finally:
        self._clear_state()
        await self.send('PARAMS 0 -1')          # Changed
```

The second command again only requires the addition of `async` and `await` anywhere coroutine behavior is required:

[Click here to view code image](#)

```
async def request_numbers(self, count):         # Changed
    for _ in range(count):
        await self.send('NUMBER')              # Changed
        data = await self.receive()            # Changed
        yield int(data)
    if self.last_distance == 0:
        return
```

The third command only requires adding one `async` and one `await` keyword:

[Click here to view code image](#)

```
async def report_outcome(self, number):         # Changed
    ...
    await self.send(f'REPORT {decision}')      # Changed
    ...
```

The code that runs the server needs to be completely reimplemented to use the asyncio built-in module and its `start_server` function:

[Click here to view code image](#)

```
import asyncio

async def handle_async_connection(reader, writer):
    session = AsyncSession(reader, writer)
    try:
        await session.loop()
    except EOFError:
        pass

async def run_async_server(address):
    server = await asyncio.start_server(
        handle_async_connection, *address)
    async with server:
        await server.serve_forever()
```

The `run_client` function that initiates the game requires changes on nearly every line. Any code that previously interacted with the blocking socket instances has to be replaced with asyncio versions of similar functionality (which are marked with `# New` below). All other lines in the function that require interaction with coroutines need to use `async` and `await` keywords as appropriate. If you forget to add one of these keywords in a necessary place, an exception will be raised at runtime.

[Click here to view code image](#)

```
async def run_async_client(address):
    streams = await asyncio.open_connection(*address)    # New
    client = AsyncClient(*streams)                       # New

    async with client.session(1, 5, 3):
        results = [(x, await client.report_outcome(x))
                   async for x in client.request_numbers(5)]

    async with client.session(10, 15, 12):
        async for number in client.request_numbers(5):
            outcome = await client.report_outcome(number)
            results.append((number, outcome))

    _, writer = streams                                # New
    writer.close()                                     # New
```



```
await writer.wait_closed() # New

return results
```

What's most interesting about `run_async_client` is that I didn't have to restructure any of the substantive parts of interacting with the `AsyncClient` in order to port this function over to use coroutines. Each of the language features that I needed has a corresponding asynchronous version, which made the migration easy to do.

This won't always be the case, though. There are currently no asynchronous versions of the `next` and `iter` built-in functions (see [Item 31: "Be Defensive When Iterating Over Arguments"](#) for background); you have to `await` on the `__anext__` and `__aiter__` methods directly. There's also no asynchronous version of `yield from` (see [Item 33: "Compose Multiple Generators with `yield from`"](#)), which makes it noisier to compose generators. But given the rapid pace at which async functionality is being added to Python, it's only a matter of time before these features become available.

Finally, the glue needs to be updated to run this new asynchronous example end-to-end. I use the `asyncio.create_task` function to enqueue the server for execution on the event loop so that it runs in parallel with the client when the `await` expression is reached. This is another approach to causing fan-out with different behavior than the `asyncio.gather` function:

[Click here to view code image](#)

```
async def main_async():
    address = ('127.0.0.1', 4321)

    server = run_async_server(address)
    asyncio.create_task(server)

    results = await run_async_client(address)
    for number, outcome in results:
        print(f'Client: {number} is {outcome}')

asyncio.run(main_async())

>>>
Guess a number between 1 and 5! Shhhhh, it's 3.
Server: 5 is Unsure
```

```
Server: 4 is Warmer
Server: 2 is Unsure
Server: 1 is Colder
Server: 3 is Correct
Guess a number between 10 and 15! Shhhhhh, it's 12.
Server: 14 is Unsure
Server: 10 is Unsure
Server: 15 is Colder
Server: 12 is Correct
Client: 5 is Unsure
Client: 4 is Warmer
Client: 2 is Unsure
Client: 1 is Colder
Client: 3 is Correct
Client: 14 is Unsure
Client: 10 is Unsure
Client: 15 is Colder
Client: 12 is Correct
```

This works as expected. The coroutine version is easier to follow because all of the interactions with threads have been removed. The `asyncio` built-in module also provides many helper functions and shortens the amount of socket boilerplate required to write a server like this.

Your use case may be more complex and harder to port for a variety of reasons. The `asyncio` module has a vast number of I/O, synchronization, and task management features that could make adopting coroutines easier for you (see [Item 62: “Mix Threads and Coroutines to Ease the Transition to `asyncio`”](#) and [Item 63: “Avoid Blocking the `asyncio` Event Loop to Maximize Responsiveness”](#)). Be sure to check out the online documentation for the library (<https://docs.python.org/3/library/asyncio.html>) to understand its full potential.

## Things to Remember

- ✦ Python provides asynchronous versions of `for` loops, `with` statements, generators, comprehensions, and library helper functions that can be used as drop-in replacements in coroutines.
- ✦ The `asyncio` built-in module makes it straightforward to port existing code that uses threads and blocking I/O over to coroutines and

asynchronous I/O.

## Item 62: Mix Threads and Coroutines to Ease the Transition to `asyncio`

In the previous item (see [Item 61: “Know How to Port Threaded I/O to `asyncio`”](#)), I ported a TCP server that does blocking I/O with threads over to use `asyncio` with coroutines. The transition was big-bang: I moved all of the code to the new style in one go. But it’s rarely feasible to port a large program this way. Instead, you usually need to incrementally migrate your codebase while also updating your tests as needed and verifying that everything works at each step along the way.

In order to do that, your codebase needs to be able to use threads for blocking I/O (see [Item 53: “Use Threads for Blocking I/O, Avoid for Parallelism”](#)) and coroutines for asynchronous I/O (see [Item 60: “Achieve Highly Concurrent I/O with Coroutines”](#)) at the same time in a way that’s mutually compatible. Practically, this means that you need threads to be able to run coroutines, and you need coroutines to be able to start and wait on threads. Luckily, `asyncio` includes built-in facilities for making this type of interoperability straightforward.

For example, say that I’m writing a program that merges log files into one output stream to aid with debugging. Given a file handle for an input log, I need a way to detect whether new data is available and return the next line of input. I can do this using the `tell` method of the file handle to check whether the current read position matches the length of the file. When no new data is present, an exception should be raised (see [Item 20: “Prefer Raising Exceptions to Returning `None`”](#) for background):

```
class NoNewData(Exception):
    pass

def readline(handle):
    offset = handle.tell()
    handle.seek(0, 2)
    length = handle.tell()

    if length == offset:
```

```
        raise NoNewData

    handle.seek(offset, 0)
    return handle.readline()
```

By wrapping this function in a while loop, I can turn it into a worker thread. When a new line is available, I call a given callback function to write it to the output log (see [Item 38: “Accept Functions Instead of Classes for Simple Interfaces”](#) for why to use a function interface for this instead of a class). When no data is available, the thread sleeps to reduce the amount of busy waiting caused by polling for new data. When the input file handle is closed, the worker thread exits:

[Click here to view code image](#)

```
import time

def tail_file(handle, interval, write_func):
    while not handle.closed:
        try:
            line = readline(handle)
        except NoNewData:
            time.sleep(interval)
        else:
            write_func(line)
```

Now, I can start one worker thread per input file and unify their output into a single output file. The write helper function below needs to use a Lock instance (see [Item 54: “Use Lock to Prevent Data Races in Threads”](#)) in order to serialize writes to the output stream and make sure that there are no intra-line conflicts:

[Click here to view code image](#)

```
from threading import Lock, Thread

def run_threads(handles, interval, output_path):
    with open(output_path, 'wb') as output:
        lock = Lock()
        def write(data):
            with lock:
                output.write(data)

        threads = []
```

```

for handle in handles:
    args = (handle, interval, write)
    thread = Thread(target=tail_file, args=args)
    thread.start()
    threads.append(thread)

for thread in threads:
    thread.join()

```

As long as an input file handle is still alive, its corresponding worker thread will also stay alive. That means it's sufficient to wait for the `join` method from each thread to complete in order to know that the whole process is done.

Given a set of input paths and an output path, I can call `run_threads` and confirm that it works as expected. How the input file handles are created or separately closed isn't important in order to demonstrate this code's behavior, nor is the output verification function—defined in `confirm_merge` that follows—which is why I've left them out here:

[Click here to view code image](#)

```

def confirm_merge(input_paths, output_path):
    ...

input_paths = ...
handles = ...
output_path = ...
run_threads(handles, 0.1, output_path)

confirm_merge(input_paths, output_path)

```

With this threaded implementation as the starting point, how can I incrementally convert this code to use `asyncio` and `coroutines` instead? There are two approaches: top-down and bottom-up.

Top-down means starting at the highest parts of a codebase, like in the main entry points, and working down to the individual functions and classes that are the leaves of the call hierarchy. This approach can be useful when you maintain a lot of common modules that you use across many different programs. By porting the entry points first, you can wait to port the common modules until you're already using `coroutines` everywhere else.

The concrete steps are:

1. Change a top function to use `async def` instead of `def`.
2. Wrap all of its calls that do I/O—potentially blocking the event loop—to use `asyncio.run_in_executor` instead.
3. Ensure that the resources or callbacks used by `run_in_executor` invocations are properly synchronized (i.e., using `Lock` or the `asyncio.run_coroutine_threadsafe` function).
4. Try to eliminate `get_event_loop` and `run_in_executor` calls by moving downward through the call hierarchy and converting intermediate functions and methods to coroutines (following the first three steps).

Here, I apply steps 1–3 to the `run_threads` function:

[Click here to view code image](#)

```
import asyncio

async def run_tasks_mixed(handles, interval, output_path):
    loop = asyncio.get_event_loop()

    with open(output_path, 'wb') as output:
        async def write_async(data):
            output.write(data)

        def write(data):
            coro = write_async(data)
            future = asyncio.run_coroutine_threadsafe(
                coro, loop)
            future.result()

        tasks = []
        for handle in handles:
            task = loop.run_in_executor(
                None, tail_file, handle, interval, write)
            tasks.append(task)

        await asyncio.gather(*tasks)
```

The `run_in_executor` method instructs the event loop to run a given function—`tail_file` in this case—using a specific `ThreadPoolExecutor` (see [Item 59: “Consider `ThreadPoolExecutor` When Threads Are Necessary for Concurrency](#)”) or a default executor instance when the first parameter is `None`. By making multiple calls to `run_in_executor` without corresponding `await` expressions, the `run_tasks_mixed` coroutine fans out to have one concurrent line of execution for each input file. Then, the `asyncio.gather` function along with an `await` expression fans in the `tail_file` threads until they all complete (see [Item 56: “Know How to Recognize When Concurrency Is Necessary](#)” for more about fan-out and fan-in).

This code eliminates the need for the `Lock` instance in the `write` helper by using `asyncio.run_coroutine_threadsafe`. This function allows plain old worker threads to call a coroutine—`write_async` in this case—and have it execute in the event loop from the main thread (or from any other thread, if necessary). This effectively synchronizes the threads together and ensures that all writes to the output file are only done by the event loop in the main thread. Once the `asyncio.gather` awaitable is resolved, I can assume that all writes to the output file have also completed, and thus I can `close` the output file handle in the `with` statement without having to worry about race conditions.

I can verify that this code works as expected. I use the `asyncio.run` function to start the coroutine and run the main event loop:

[Click here to view code image](#)

```
input_paths = ...
handles = ...
output_path = ...
asyncio.run(run_tasks_mixed(handles, 0.1, output_path))

confirm_merge(input_paths, output_path)
```

Now, I can apply step 4 to the `run_tasks_mixed` function by moving down the call stack. I can redefine the `tail_file` dependent function to be an asynchronous coroutine instead of doing blocking I/O by following steps 1–3:

[Click here to view code image](#)

```
async def tail_async(handle, interval, write_func):
    loop = asyncio.get_event_loop()

    while not handle.closed:
        try:
            line = await loop.run_in_executor(
                None, readline, handle)
        except NoNewData:
            await asyncio.sleep(interval)
        else:
            await write_func(line)
```

This new implementation of `tail_async` allows me to push calls to `get_event_loop` and `run_in_executor` down the stack and out of the `run_tasks_mixed` function entirely. What's left is clean and much easier to follow:

[Click here to view code image](#)

```
async def run_tasks(handles, interval, output_path):
    with open(output_path, 'wb') as output:
        async def write_async(data):
            output.write(data)
        tasks = []
        for handle in handles:
            coro = tail_async(handle, interval, write_async)
            task = asyncio.create_task(coro)
            tasks.append(task)
        await asyncio.gather(*tasks)
```

I can verify that `run_tasks` works as expected, too:

[Click here to view code image](#)

```
input_paths = ...
handles = ...
output_path = ...
asyncio.run(run_tasks(handles, 0.1, output_path))

confirm_merge(input_paths, output_path)
```

It's possible to continue this iterative refactoring pattern and convert `readline` into an asynchronous coroutine as well. However, that function requires so many blocking file I/O operations that it doesn't seem worth



porting, given how much that would reduce the clarity of the code and hurt performance. In some situations, it makes sense to move everything to `asyncio`, and in others it doesn't.

The bottom-up approach to adopting coroutines has four steps that are similar to the steps of the top-down style, but the process traverses the call hierarchy in the opposite direction: from leaves to entry points.

The concrete steps are:

1. Create a new asynchronous coroutine version of each leaf function that you're trying to port.
2. Change the existing synchronous functions so they call the coroutine versions and run the event loop instead of implementing any real behavior.
3. Move up a level of the call hierarchy, make another layer of coroutines, and replace existing calls to synchronous functions with calls to the coroutines defined in step 1.
4. Delete synchronous wrappers around coroutines created in step 2 as you stop requiring them to glue the pieces together.

For the example above, I would start with the `tail_file` function since I decided that the `readline` function should keep using blocking I/O. I can rewrite `tail_file` so it merely wraps the `tail_async` coroutine that I defined above. To run that coroutine until it finishes, I need to create an event loop for each `tail_file` worker thread and then call its `run_until_complete` method. This method will block the current thread and drive the event loop until the `tail_async` coroutine exits, achieving the same behavior as the threaded, blocking I/O version of `tail_file`:

[Click here to view code image](#)

```
def tail_file(handle, interval, write_func):  
    loop = asyncio.new_event_loop()  
    asyncio.set_event_loop(loop)
```

```

async def write_async(data):
    write_func(data)

coro = tail_async(handle, interval, write_async)
loop.run_until_complete(coro)

```

This new `tail_file` function is a drop-in replacement for the old one. I can verify that everything works as expected by calling `run_threads` again:

[Click here to view code image](#)

```

input_paths = ...
handles = ...
output_path = ...
run_threads(handles, 0.1, output_path)

confirm_merge(input_paths, output_path)

```

After wrapping `tail_async` with `tail_file`, the next step is to convert the `run_threads` function to a coroutine. This ends up being the same work as step 4 of the top-down approach above, so at this point, the styles converge.

This is a great start for adopting `asyncio`, but there's even more that you could do to increase the responsiveness of your program (see [Item 63: “Avoid Blocking the `asyncio` Event Loop to Maximize Responsiveness](#)”).

## Things to Remember

- ✦ The awaitable `run_in_executor` method of the `asyncio` event loop enables coroutines to run synchronous functions in `ThreadPoolExecutor` pools. This facilitates top-down migrations to `asyncio`.
- ✦ The `run_until_complete` method of the `asyncio` event loop enables synchronous code to run a coroutine until it finishes. The `asyncio.run_coroutine_threadsafe` function provides the same functionality across thread boundaries. Together these help with bottom-up migrations to `asyncio`.

## Item 63: Avoid Blocking the `asyncio` Event Loop to Maximize Responsiveness

In the previous item I showed how to migrate to `asyncio` incrementally (see [Item 62: “Mix Threads and Coroutines to Ease the Transition to `asyncio`”](#) for background and the implementation of various functions below). The resulting coroutine properly tails input files and merges them into a single output:

[Click here to view code image](#)

```
import asyncio

async def run_tasks(handles, interval, output_path):
    with open(output_path, 'wb') as output:
        async def write_async(data):
            output.write(data)

        tasks = []
        for handle in handles:
            coro = tail_async(handle, interval, write_async)
            task = asyncio.create_task(coro)
            tasks.append(task)

        await asyncio.gather(*tasks)
```

However, it still has one big problem: The `open`, `close`, and `write` calls for the output file handle happen in the main event loop. These operations all require making system calls to the program’s host operating system, which may block the event loop for significant amounts of time and prevent other coroutines from making progress. This could hurt overall responsiveness and increase latency, especially for programs such as highly concurrent servers.

I can detect when this problem happens by passing the `debug=True` parameter to the `asyncio.run` function. Here, I show how the file and line of a bad coroutine, presumably blocked on a slow system call, can be identified:

[Click here to view code image](#)

```
import time

async def slow_coroutine():
    time.sleep(0.5) # Simulating slow I/O

asyncio.run(slow_coroutine(), debug=True)

>>>
Executing <Task finished name='Task-1' coro=<slow_coroutine()
➡done, defined at example.py:29> result=None created
➡at .../asyncio/base_events.py:487> took 0.503 seconds
...
```

If I want the most responsive program possible, I need to minimize the potential system calls that are made from within the event loop. In this case, I can create a new Thread subclass (see [Item 53: “Use Threads for Blocking I/O, Avoid for Parallelism”](#)) that encapsulates everything required to write to the output file using its own event loop:

[Click here to view code image](#)

```
from threading import Thread

class WriteThread(Thread):
    def __init__(self, output_path):
        super().__init__()
        self.output_path = output_path
        self.output = None
        self.loop = asyncio.new_event_loop()

    def run(self):
        asyncio.set_event_loop(self.loop)
        with open(self.output_path, 'wb') as self.output:
            self.loop.run_forever()

        # Run one final round of callbacks so the await on
        # stop() in another event loop will be resolved.
        self.loop.run_until_complete(asyncio.sleep(0))
```

Coroutines in other threads can directly call and await on the write method of this class, since it’s merely a thread-safe wrapper around the real\_write method that actually does the I/O. This eliminates the need for a Lock (see [Item 54: “Use Lock to Prevent Data Races in Threads”](#)):

[Click here to view code image](#)

```

async def real_write(self, data):
    self.output.write(data)

async def write(self, data):
    coro = self.real_write(data)
    future = asyncio.run_coroutine_threadsafe(
        coro, self.loop)
    await asyncio.wrap_future(future)

```

Other coroutines can tell the worker thread when to stop in a threadsafe manner, using similar boilerplate:

[Click here to view code image](#)

```

async def real_stop(self):
    self.loop.stop()
async def stop(self):
    coro = self.real_stop()
    future = asyncio.run_coroutine_threadsafe(
        coro, self.loop)
    await asyncio.wrap_future(future)

```

I can also define the `__aenter__` and `__aexit__` methods to allow this class to be used in with statements (see [Item 66: “Consider contextlib and with Statements for Reusable try/finally Behavior”](#)). This ensures that the worker thread starts and stops at the right times without slowing down the main event loop thread:

[Click here to view code image](#)

```

async def __aenter__(self):
    loop = asyncio.get_event_loop()
    await loop.run_in_executor(None, self.start)
    return self

async def __aexit__(self, *_):
    await self.stop()

```

With this new `WriteThread` class, I can refactor `run_tasks` into a fully asynchronous version that’s easy to read and completely avoids running slow system calls in the main event loop thread:

[Click here to view code image](#)

```
def readline(handle):
    ...

async def tail_async(handle, interval, write_func):
    ...

async def run_fully_async(handles, interval, output_path):
    async with WriteThread(output_path) as output:
        tasks = []
        for handle in handles:
            coro = tail_async(handle, interval, output.write)
            task = asyncio.create_task(coro)
            tasks.append(task)

        await asyncio.gather(*tasks)
```

I can verify that this works as expected, given a set of input handles and an output file path:

[Click here to view code image](#)

```
def confirm_merge(input_paths, output_path):
    ...
input_paths = ...
handles = ...
output_path = ...
asyncio.run(run_fully_async(handles, 0.1, output_path))

confirm_merge(input_paths, output_path)
```

## Things to Remember

- ✦ Making system calls in coroutines—including blocking I/O and starting threads—can reduce program responsiveness and increase the perception of latency.
- ✦ Pass the `debug=True` parameter to `asyncio.run` in order to detect when certain coroutines are preventing the event loop from reacting quickly.

## Item 64: Consider `concurrent.futures` for True Parallelism

At some point in writing Python programs, you may hit the performance wall. Even after optimizing your code (see [Item 70: “Profile Before](#)

Optimizing”), your program’s execution may still be too slow for your needs. On modern computers that have an increasing number of CPU cores, it’s reasonable to assume that one solution would be parallelism. What if you could split your code’s computation into independent pieces of work that run simultaneously across multiple CPU cores?

Unfortunately, Python’s global interpreter lock (GIL) prevents true parallelism in threads (see [Item 53: “Use Threads for Blocking I/O, Avoid for Parallelism”](#)), so that option is out. Another common suggestion is to rewrite your most performance-critical code as an extension module, using the C language. C gets you closer to the bare metal and can run faster than Python, eliminating the need for parallelism in some cases. C extensions can also start native threads independent of the Python interpreter that run in parallel and utilize multiple CPU cores with no concern for the GIL. Python’s API for C extensions is well documented and a good choice for an escape hatch. It’s also worth checking out tools like SWIG (<https://github.com/swig/swig>) and CLIF (<https://github.com/google/clif>) to aid in extension development.

But rewriting your code in C has a high cost. Code that is short and understandable in Python can become verbose and complicated in C. Such a port requires extensive testing to ensure that the functionality is equivalent to the original Python code and that no bugs have been introduced. Sometimes it’s worth it, which explains the large ecosystem of C-extension modules in the Python community that speed up things like text parsing, image compositing, and matrix math. There are even open source tools such as Cython (<https://cython.org>) and Numba (<https://numba.pydata.org>) that can ease the transition to C.

The problem is that moving one piece of your program to C isn’t sufficient most of the time. Optimized Python programs usually don’t have one major source of slowness; rather, there are often many significant contributors. To get the benefits of C’s bare metal and threads, you’d need to port large parts of your program, drastically increasing testing needs and risk. There must be a better way to preserve your investment in Python to solve difficult computational problems.

The multiprocessing built-in module, which is easily accessed via the `concurrent.futures` built-in module, may be exactly what you need (see [Item 59: “Consider `ThreadPoolExecutor` When Threads Are Necessary for Concurrency](#)” for a related example). It enables Python to utilize multiple CPU cores in parallel by running additional interpreters as child processes. These child processes are separate from the main interpreter, so their global interpreter locks are also separate. Each child can fully utilize one CPU core. Each child has a link to the main process where it receives instructions to do computation and returns results.

For example, say that I want to do something computationally intensive with Python and utilize multiple CPU cores. I’ll use an implementation of finding the greatest common divisor of two numbers as a proxy for a more computationally intense algorithm (like simulating fluid dynamics with the Navier–Stokes equation):

[Click here to view code image](#)

```
# my_module.py
def gcd(pair):
    a, b = pair
    low = min(a, b)
    for i in range(low, 0, -1):
        if a % i == 0 and b % i == 0:
            return i
    assert False, 'Not reachable'
```

Running this function in serial takes a linearly increasing amount of time because there is no parallelism:

[Click here to view code image](#)

```
# run_serial.py
import my_module
import time

NUMBERS = [
    (1963309, 2265973), (2030677, 3814172),
    (1551645, 2229620), (2039045, 2020802),
    (1823712, 1924928), (2293129, 1020491),
    (1281238, 2273782), (3823812, 4237281),
    (3812741, 4729139), (1292391, 2123811),
]
```



```
def main():
    start = time.time()
    results = list(map(my_module.gcd, NUMBERS))
    end = time.time()
    delta = end - start
    print(f'Took {delta:.3f} seconds')

if __name__ == '__main__':
    main()
```

```
>>>
Took 1.173 seconds
```

Running this code on multiple Python threads will yield no speed improvement because the GIL prevents Python from using multiple CPU cores in parallel. Here, I do the same computation as above but using the `concurrent.futures` module with its `ThreadPoolExecutor` class and two worker threads (to match the number of CPU cores on my computer):

[Click here to view code image](#)

```
# run_threads.py
import my_module
from concurrent.futures import ThreadPoolExecutor
import time

NUMBERS = [
    ...
]

def main():
    start = time.time()
    pool = ThreadPoolExecutor(max_workers=2)
    results = list(pool.map(my_module.gcd, NUMBERS))
    end = time.time()
    delta = end - start
    print(f'Took {delta:.3f} seconds')

if __name__ == '__main__':
    main()

>>>
Took 1.436 seconds
```

It's even slower this time because of the overhead of starting and communicating with the pool of threads.

Now for the surprising part: Changing a single line of code causes something magical to happen. If I replace the `ThreadPoolExecutor` with the `ProcessPoolExecutor` from the `concurrent.futures` module, everything speeds up:

[Click here to view code image](#)

```
# run_parallel.py
import my_module
from concurrent.futures import ProcessPoolExecutor
import time

NUMBERS = [
    ...
]

def main():
    start = time.time()
    pool = ProcessPoolExecutor(max_workers=2) # The one change
    results = list(pool.map(my_module.gcd, NUMBERS))
    end = time.time()
    delta = end - start
    print(f'Took {delta:.3f} seconds')

if __name__ == '__main__':
    main()

>>>
Took 0.683 seconds
```

Running on my dual-core machine, this is significantly faster! How is this possible? Here's what the `ProcessPoolExecutor` class actually does (via the low-level constructs provided by the `multiprocessing` module):

1. It takes each item from the `numbers` input data to `map`.
2. It serializes the item into binary data by using the `pickle` module (see [Item 68: “Make pickle Reliable with copyreg”](#)).

3. It copies the serialized data from the main interpreter process to a child interpreter process over a local socket.
4. It deserializes the data back into Python objects, using `pickle` in the child process.
5. It imports the Python module containing the `gcd` function.
6. It runs the function on the input data in parallel with other child processes.
7. It serializes the result back into binary data.
8. It copies that binary data back through the socket.
9. It deserializes the binary data back into Python objects in the parent process.
10. It merges the results from multiple children into a single `list` to return.

Although it looks simple to the programmer, the `multiprocessing` module and `ProcessPoolExecutor` class do a huge amount of work to make parallelism possible. In most other languages, the only touch point you need to coordinate two threads is a single lock or atomic operation (see [Item 54: “Use Lock to Prevent Data Races in Threads”](#) for an example). The overhead of using `multiprocessing` via `ProcessPoolExecutor` is high because of all of the serialization and deserialization that must happen between the parent and child processes.

This scheme is well suited to certain types of isolated, high-leverage tasks. By *isolated*, I mean functions that don’t need to share state with other parts of the program. By *high-leverage tasks*, I mean situations in which only a small amount of data must be transferred between the parent and child processes to enable a large amount of computation. The greatest common divisor algorithm is one example of this, but many other mathematical algorithms work similarly.

If your computation doesn't have these characteristics, then the overhead of `ProcessPoolExecutor` may prevent it from speeding up your program through parallelization. When that happens, `multiprocessing` provides more advanced facilities for shared memory, cross-process locks, queues, and proxies. But all of these features are very complex. It's hard enough to reason about such tools in the memory space of a single process shared between Python threads. Extending that complexity to other processes and involving sockets makes this much more difficult to understand.

I suggest that you initially avoid all parts of the `multiprocessing` built-in module. You can start by using the `ThreadPoolExecutor` class to run isolated, high-leverage functions in threads. Later you can move to the `ProcessPoolExecutor` to get a speedup. Finally, when you've completely exhausted the other options, you can consider using the `multiprocessing` module directly.

## Things to Remember

- ◆ Moving CPU bottlenecks to C-extension modules can be an effective way to improve performance while maximizing your investment in Python code. However, doing so has a high cost and may introduce bugs.
- ◆ The `multiprocessing` module provides powerful tools that can parallelize certain types of Python computation with minimal effort.
- ◆ The power of `multiprocessing` is best accessed through the `concurrent.futures` built-in module and its simple `ProcessPoolExecutor` class.
- ◆ Avoid the advanced (and complicated) parts of the `multiprocessing` module until you've exhausted all other options.

## 8. Robustness and Performance

Once you’ve written a useful Python program, the next step is to *productionize* your code so it’s bulletproof. Making programs dependable when they encounter unexpected circumstances is just as important as making programs with correct functionality. Python has built-in features and modules that aid in hardening your programs so they are robust in a wide variety of situations.

One dimension of robustness is scalability and performance. When you’re implementing Python programs that handle a non-trivial amount of data, you’ll often see slowdowns caused by the algorithmic complexity of your code or other types of computational overhead. Luckily, Python includes many of the algorithms and data structures you need to achieve high performance with minimal effort.

### Item 65: Take Advantage of Each Block in

`try/except/else/finally`

There are four distinct times when you might want to take action during exception handling in Python. These are captured in the functionality of `try`, `except`, `else`, and `finally` blocks. Each block serves a unique purpose in the compound statement, and their various combinations are useful (see [Item 87: “Define a Root Exception to Insulate Callers from APIs”](#) for another example).

#### `finally` Blocks

Use `try/finally` when you want exceptions to propagate up but also want to run cleanup code even when exceptions occur. One common usage of `try/finally` is for reliably closing file handles (see [Item 66: “Consider `contextlib` and `with` Statements for Reusable `try/finally` Behavior”](#) for another—likely better—approach):

[Click here to view code image](#)

```
def try_finally_example(filename):
    print('* Opening file')
    handle = open(filename, encoding='utf-8') # Maybe OSError
    try:
        print('* Reading data')
        return handle.read() # Maybe UnicodeDecodeError
    finally:
        print('* Calling close()')
        handle.close() # Always runs after try block
```

Any exception raised by the read method will always propagate up to the calling code, but the close method of handle in the finally block will run first:

[Click here to view code image](#)

```
filename = 'random_data.txt'

with open(filename, 'wb') as f:
    f.write(b'\xf1\xf2\xf3\xf4\xf5') # Invalid utf-8

data = try_finally_example(filename)
```

```
>>>
* Opening file
* Reading data
* Calling close()
Traceback ...
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xf1 in
➔position 0: invalid continuation byte
```

You must call open before the try block because exceptions that occur when opening the file (like OSError if the file does not exist) should skip the finally block entirely:

[Click here to view code image](#)

```
try_finally_example('does_not_exist.txt')

>>>
* Opening file
Traceback ...
FileNotFoundError: [Errno 2] No such file or directory:
➔'does_not_exist.txt'
```

## else Blocks

Use try/except/else to make it clear which exceptions will be handled by your code and which exceptions will propagate up. When the try block doesn't raise an exception, the else block runs. The else block helps you minimize the amount of code in the try block, which is good for isolating potential exception causes and improves readability. For example, say that I want to load JSON dictionary data from a string and return the value of a key it contains:

[Click here to view code image](#)

```
import json

def load_json_key(data, key):
    try:
        print('* Loading JSON data')
        result_dict = json.loads(data) # May raise ValueError
    except ValueError as e:
        print('* Handling ValueError')
        raise KeyError(key) from e
    else:
        print('* Looking up key')
        return result_dict[key] # May raise KeyError
```

In the successful case, the JSON data is decoded in the try block, and then the key lookup occurs in the else block:

[Click here to view code image](#)

```
assert load_json_key('{"foo": "bar"}', 'foo') == 'bar'

>>>
* Loading JSON data
* Looking up key
```

If the input data isn't valid JSON, then decoding with `json.loads` raises a `ValueError`. The exception is caught by the except block and handled:

[Click here to view code image](#)

```
load_json_key('{"foo": bad payload', 'foo')

>>>
```

```
* Loading JSON data
* Handling ValueError
Traceback ...
JSONDecodeError: Expecting value: line 1 column 9 (char 8)
```

The above exception was the direct cause of the following  
➡exception:

```
Traceback ...
KeyError: 'foo'
```

If the key lookup raises any exceptions, they propagate up to the caller because they are outside the try block. The else clause ensures that what follows the try/except is visually distinguished from the except block. This makes the exception propagation behavior clear:

[Click here to view code image](#)

```
load_json_key({'foo': 'bar'}, 'does not exist')
>>>
* Loading JSON data
* Looking up key
Traceback ...
KeyError: 'does not exist'
```

## Everything Together

Use try/except/else/finally when you want to do it all in one compound statement. For example, say that I want to read a description of work to do from a file, process it, and then update the file in-place. Here, the try block is used to read the file and process it; the except block is used to handle exceptions from the try block that are expected; the else block is used to update the file in place and allow related exceptions to propagate up; and the finally block cleans up the file handle:

[Click here to view code image](#)

```
UNDEFINED = object()

def divide_json(path):
    print('* Opening file')
    handle = open(path, 'r+')    # May raise OSError
    try:
        print('* Reading data')
```



```

data = handle.read()    # May raise UnicodeDecodeError
print('* Loading JSON data')
op = json.loads(data)    # May raise ValueError
print('* Performing calculation')
value = (
    op['numerator'] /
    op['denominator']) # May raise ZeroDivisionError
except ZeroDivisionError as e:
    print('* Handling ZeroDivisionError')
    return UNDEFINED
else:
    print('* Writing calculation')
    op['result'] = value
    result = json.dumps(op)
    handle.seek(0)        # May raise OSError
    handle.write(result)   # May raise OSError
    return value
finally:
    print('* Calling close()')
    handle.close()        # Always runs

```

In the successful case, the try, else, and finally blocks run:

[Click here to view code image](#)

```

temp_path = 'random_data.json'

with open(temp_path, 'w') as f:
    f.write('{"numerator": 1, "denominator": 10}')

assert divide_json(temp_path) == 0.1

>>>
* Opening file
* Reading data
* Loading JSON data
* Performing calculation
* Writing calculation
* Calling close()

```

If the calculation is invalid, the try, except, and finally blocks run, but the else block does not:

[Click here to view code image](#)

```

with open(temp_path, 'w') as f:
    f.write('{"numerator": 1, "denominator": 0}')

```

```
assert divide_json(temp_path) is UNDEFINED
```

```
>>>
```

```
* Opening file
* Reading data
* Loading JSON data
* Performing calculation
* Handling ZeroDivisionError
* Calling close()
```

If the JSON data was invalid, the try block runs and raises an exception, the finally block runs, and then the exception is propagated up to the caller. The except and else blocks do not run:

[Click here to view code image](#)

```
with open(temp_path, 'w') as f:
    f.write('{"numerator": 1 bad data')
```

```
divide_json(temp_path)
```

```
>>>
```

```
* Opening file
* Reading data
* Loading JSON data
* Calling close()
```

```
Traceback ...
```

```
JSONDecodeError: Expecting ',' delimiter: line 1 column 17
```

```
➔(char 16)
```

This layout is especially useful because all of the blocks work together in intuitive ways. For example, here I simulate this by running the divide\_json function at the same time that my hard drive runs out of disk space:

[Click here to view code image](#)

```
with open(temp_path, 'w') as f:
    f.write('{"numerator": 1, "denominator": 10}')
```

```
divide_json(temp_path)
```

```
>>>
```

```
* Opening file
* Reading data
```

```
* Loading JSON data
* Performing calculation
* Writing calculation
* Calling close()
Traceback ...
OSError: [Errno 28] No space left on device
```

When the exception was raised in the `else` block while rewriting the result data, the `finally` block still ran and closed the file handle as expected.

## Things to Remember

- ◆ The `try/finally` compound statement lets you run cleanup code regardless of whether exceptions were raised in the `try` block.
- ◆ The `else` block helps you minimize the amount of code in `try` blocks and visually distinguish the success case from the `try/except` blocks.
- ◆ An `else` block can be used to perform additional actions after a successful `try` block but before common cleanup in a `finally` block.

## Item 66: Consider `contextlib` and `with` Statements for Reusable `try/finally` Behavior

The `with` statement in Python is used to indicate when code is running in a special context. For example, mutual-exclusion locks (see [Item 54: “Use Lock to Prevent Data Races in Threads”](#)) can be used in `with` statements to indicate that the indented code block runs only while the lock is held:

[Click here to view code image](#)

```
from threading import Lock

lock = Lock()

with lock:
    # Do something while maintaining an invariant
    ...
```

The example above is equivalent to this `try/finally` construction because the `Lock` class properly enables the `with` statement (see [Item 65: “Take](#)

[Advantage of Each Block in try/except/else/finally](#)” for more about try/finally):

[Click here to view code image](#)

```
lock.acquire()
try:
    # Do something while maintaining an invariant
    ...
finally:
    lock.release()
```

The with statement version of this is better because it eliminates the need to write the repetitive code of the try/finally construction, and it ensures that you don’t forget to have a corresponding release call for every acquire call.

It’s easy to make your objects and functions work in with statements by using the contextlib built-in module. This module contains the contextmanager decorator (see [Item 26: “Define Function Decorators with functools.wraps”](#) for background), which lets a simple function be used in with statements. This is much easier than defining a new class with the special methods `__enter__` and `__exit__` (the standard way).

For example, say that I want a region of code to have more debug logging sometimes. Here, I define a function that does logging at two severity levels:

[Click here to view code image](#)

```
import logging

def my_function():
    logging.debug('Some debug data')
    logging.error('Error log here')
    logging.debug('More debug data')
```

The default log level for my program is WARNING, so only the error message will print to screen when I run the function:

```
my_function()
```

```
>>>
Error log here
```

I can elevate the log level of this function temporarily by defining a context manager. This helper function boosts the logging severity level before running the code in the `with` block and reduces the logging severity level afterward:

[Click here to view code image](#)

```
from contextlib import contextmanager

@contextmanager
def debug_logging(level):
    logger = logging.getLogger()
    old_level = logger.getEffectiveLevel()
    logger.setLevel(level)
    try:
        yield
    finally:
        logger.setLevel(old_level)
```

The `yield` expression is the point at which the `with` block's contents will execute (see [Item 30: “Consider Generators Instead of Returning Lists”](#) for background). Any exceptions that happen in the `with` block will be re-raised by the `yield` expression for you to catch in the helper function (see [Item 35: “Avoid Causing State Transitions in Generators with `throw`”](#) for how that works).

Now, I can call the same logging function again but in the `debug_logging` context. This time, all of the debug messages are printed to the screen during the `with` block. The same function running outside the `with` block won't print debug messages:

```
with debug_logging(logging.DEBUG):
    print('* Inside:')
    my_function()

print('* After:')
my_function()

>>>
* Inside:
Some debug data
```

```
Error log here
More debug data
* After:
Error log here
```

## Using `with` Targets

The context manager passed to a `with` statement may also return an object. This object is assigned to a local variable in the `as` part of the compound statement. This gives the code running in the `with` block the ability to directly interact with its context.

For example, say I want to write a file and ensure that it's always closed correctly. I can do this by passing `open` to the `with` statement. `open` returns a file handle for the `as` target of `with`, and it closes the handle when the `with` block exits:

[Click here to view code image](#)

```
with open('my_output.txt', 'w') as handle:
    handle.write('This is some data!')
```

This approach is more Pythonic than manually opening and closing the file handle every time. It gives you confidence that the file is eventually closed when execution leaves the `with` statement. By highlighting the critical section, it also encourages you to reduce the amount of code that executes while the file handle is open, which is good practice in general.

To enable your own functions to supply values for `as` targets, all you need to do is `yield` a value from your context manager. For example, here I define a context manager to fetch a `Logger` instance, set its level, and then `yield` it as the target:

[Click here to view code image](#)

```
@contextmanager
def log_level(level, name):
    logger = logging.getLogger(name)
    old_level = logger.getEffectiveLevel()
    logger.setLevel(level)
    try:
        yield logger
```

```
finally:
    logger.setLevel(old_level)
```

Calling logging methods like `debug` on the `as` target produces output because the logging severity level is set low enough in the `with` block on that specific `Logger` instance. Using the `logging` module directly won't print anything because the default logging severity level for the default program logger is `WARNING`:

[Click here to view code image](#)

```
with log_level(logging.DEBUG, 'my-log') as logger:
    logger.debug(f'This is a message for {logger.name}!')
    logging.debug('This will not print')
```

```
>>>
This is a message for my-log!
```

After the `with` statement exits, calling `debug` logging methods on the `Logger` named `'my-log'` will not print anything because the default logging severity level has been restored. Error log messages will always print:

[Click here to view code image](#)

```
logger = logging.getLogger('my-log')
logger.debug('Debug will not print')
logger.error('Error will print')
```

```
>>>
Error will print
```

Later, I can change the name of the logger I want to use by simply updating the `with` statement. This will point the `Logger` that's the `as` target in the `with` block to a different instance, but I won't have to update any of my other code to match:

[Click here to view code image](#)

```
with log_level(logging.DEBUG, 'other-log') as logger:
    logger.debug(f'This is a message for {logger.name}!')
    logging.debug('This will not print')
```

```
>>>
This is a message for other-log!
```

This isolation of state and decoupling between creating a context and acting within that context is another benefit of the `with` statement.

## Things to Remember

- ✦ The `with` statement allows you to reuse logic from `try/finally` blocks and reduce visual noise.
- ✦ The `contextlib` built-in module provides a `contextmanager` decorator that makes it easy to use your own functions in `with` statements.
- ✦ The value yielded by context managers is supplied to the `as` part of the `with` statement. It's useful for letting your code directly access the cause of a special context.

## Item 67: Use `datetime` Instead of `time` for Local Clocks

Coordinated Universal Time (UTC) is the standard, time-zone-independent representation of time. UTC works great for computers that represent time as seconds since the UNIX epoch. But UTC isn't ideal for humans. Humans reference time relative to where they're currently located. People say "noon" or "8 am" instead of "UTC 15:00 minus 7 hours." If your program handles time, you'll probably find yourself converting time between UTC and local clocks for the sake of human understanding.

Python provides two ways of accomplishing time zone conversions. The old way, using the `time` built-in module, is terribly error prone. The new way, using the `datetime` built-in module, works great with some help from the community-built package named `pytz`.

You should be acquainted with both `time` and `datetime` to thoroughly understand why `datetime` is the best choice and `time` should be avoided.

### The `time` Module

The `localtime` function from the `time` built-in module lets you convert a UNIX timestamp (seconds since the UNIX epoch in UTC) to a local time that matches the host computer's time zone (Pacific Daylight Time in my



case). This local time can be printed in human-readable format using the `strftime` function:

[Click here to view code image](#)

```
import time

now = 1552774475
local_tuple = time.localtime(now)
time_format = '%Y-%m-%d %H:%M:%S'
time_str = time.strftime(time_format, local_tuple)
print(time_str)

>>>
2019-03-16 15:14:35
```

You'll often need to go the other way as well, starting with user input in human-readable local time and converting it to UTC time. You can do this by using the `strptime` function to parse the time string, and then calling `mktime` to convert local time to a UNIX timestamp:

[Click here to view code image](#)

```
time_tuple = time.strptime(time_str, time_format)
utc_now = time.mktime(time_tuple)
print(utc_now)

>>>
1552774475.0
```

How do you convert local time in one time zone to local time in another time zone? For example, say that I'm taking a flight between San Francisco and New York, and I want to know what time it will be in San Francisco when I've arrived in New York.

I might initially assume that I can directly manipulate the return values from the `time`, `localtime`, and `strptime` functions to do time zone conversions. But this is a very bad idea. Time zones change all the time due to local laws. It's too complicated to manage yourself, especially if you want to handle every global city for flight departures and arrivals.

Many operating systems have configuration files that keep up with the time zone changes automatically. Python lets you use these time zones through

the `time` module if your platform supports it. On other platforms, such as Windows, some time zone functionality isn't available from `time` at all. For example, here I parse a departure time from the San Francisco time zone, Pacific Daylight Time (PDT):

[Click here to view code image](#)

```
import os

if os.name == 'nt':
    print("This example doesn't work on Windows")
else:
    parse_format = '%Y-%m-%d %H:%M:%S %Z'
    depart_sfo = '2019-03-16 15:45:16 PDT'
    time_tuple = time.strptime(depart_sfo, parse_format)
    time_str = time.strftime(time_format, time_tuple)
    print(time_str)

>>>
2019-03-16 15:45:16
```

After seeing that 'PDT' works with the `strptime` function, I might also assume that other time zones known to my computer will work. Unfortunately, this isn't the case. `strptime` raises an exception when it sees Eastern Daylight Time (EDT), which is the time zone for New York:

[Click here to view code image](#)

```
arrival_nyc = '2019-03-16 23:33:24 EDT'
time_tuple = time.strptime(arrival_nyc, time_format)

>>>
Traceback ...
ValueError: unconverted data remains:  EDT
```

The problem here is the platform-dependent nature of the `time` module. Its behavior is determined by how the underlying C functions work with the host operating system. This makes the functionality of the `time` module unreliable in Python. The `time` module fails to consistently work properly for multiple local times. Thus, you should avoid using the `time` module for this purpose. If you must use `time`, use it only to convert between UTC and

the host computer's local time. For all other types of conversions, use the `datetime` module.

## The `datetime` Module

The second option for representing times in Python is the `datetime` class from the `datetime` built-in module. Like the `time` module, `datetime` can be used to convert from the current time in UTC to local time.

Here, I convert the present time in UTC to my computer's local time, PDT:

[Click here to view code image](#)

```
from datetime import datetime, timezone

now = datetime(2019, 3, 16, 22, 14, 35)
now_utc = now.replace(tzinfo=timezone.utc)
now_local = now_utc.astimezone()
print(now_local)

>>>
2019-03-16 15:14:35-07:00
```

The `datetime` module can also easily convert a local time back to a UNIX timestamp in UTC:

[Click here to view code image](#)

```
time_str = '2019-03-16 15:14:35'
now = datetime.strptime(time_str, time_format)
time_tuple = now.timetuple()
utc_now = time.mktime(time_tuple)
print(utc_now)

>>>
1552774475.0
```

Unlike the `time` module, the `datetime` module has facilities for reliably converting from one local time to another local time. However, `datetime` only provides the machinery for time zone operations with its `tzinfo` class and related methods. The Python default installation is missing time zone definitions besides UTC.

Luckily, the Python community has addressed this gap with the `pytz` module that's available for download from the Python Package Index (see [Item 82: “Know Where to Find Community-Built Modules”](#) for how to install it). `pytz` contains a full database of every time zone definition you might need.

To use `pytz` effectively, you should always convert local times to UTC first. Perform any datetime operations you need on the UTC values (such as offsetting). Then, convert to local times as a final step.

For example, here I convert a New York City flight arrival time to a UTC datetime. Although some of these calls seem redundant, all of them are necessary when using `pytz`:

[Click here to view code image](#)

```
import pytz

arrival_nyc = '2019-03-16 23:33:24'
nyc_dt_naive = datetime.strptime(arrival_nyc, time_format)
eastern = pytz.timezone('US/Eastern')
nyc_dt = eastern.localize(nyc_dt_naive)
utc_dt = pytz.utc.normalize(nyc_dt.astimezone(pytz.utc))
print(utc_dt)
```

```
>>>
2019-03-17 03:33:24+00:00
```

Once I have a UTC datetime, I can convert it to San Francisco local time:

[Click here to view code image](#)

```
pacific = pytz.timezone('US/Pacific')
sf_dt = pacific.normalize(utc_dt.astimezone(pacific))
print(sf_dt)
```

```
>>>
2019-03-16 20:33:24-07:00
```

Just as easily, I can convert it to the local time in Nepal:

[Click here to view code image](#)

```
nepal = pytz.timezone('Asia/Katmandu')
nepal_dt = nepal.normalize(utc_dt.astimezone(nepal))
print(nepal_dt)

>>>
2019-03-17 09:18:24+05:45
```

With `datetime` and `pytz`, these conversions are consistent across all environments, regardless of what operating system the host computer is running.

## Things to Remember

- ✦ Avoid using the `time` module for translating between different time zones.
- ✦ Use the `datetime` built-in module along with the `pytz` community module to reliably convert between times in different time zones.
- ✦ Always represent time in UTC and do conversions to local time as the very final step before presentation.

## Item 68: Make `pickle` Reliable with `copyreg`

The `pickle` built-in module can serialize Python objects into a stream of bytes and deserialize bytes back into objects. Pickled byte streams shouldn't be used to communicate between untrusted parties. The purpose of `pickle` is to let you pass Python objects between programs that you control over binary channels.

### Note

The `pickle` module's serialization format is unsafe by design. The serialized data contains what is essentially a program that describes how to reconstruct the original Python object. This means a malicious `pickle` payload could be used to compromise any part of a Python program that attempts to deserialize it.

In contrast, the `json` module is safe by design. Serialized JSON data contains a simple description of an object hierarchy. Deserializing JSON data does not expose a Python program to additional risk. Formats like JSON should be used for communication between programs or people who don't trust each other.

For example, say that I want to use a Python object to represent the state of a player's progress in a game. The game state includes the level the player is on and the number of lives they have remaining:

```
class GameState:
    def __init__(self):
        self.level = 0
        self.lives = 4
```

The program modifies this object as the game runs:

[Click here to view code image](#)

```
state = GameState()
state.level += 1 # Player beat a level
state.lives -= 1 # Player had to try again

print(state.__dict__)

>>>
{'level': 1, 'lives': 3}
```

When the user quits playing, the program can save the state of the game to a file so it can be resumed at a later time. The `pickle` module makes it easy to do this. Here, I use the `dump` function to write the `GameState` object to a file:

```
import pickle

state_path = 'game_state.bin'
with open(state_path, 'wb') as f:
    pickle.dump(state, f)
```

Later, I can call the `load` function with the file and get back the `GameState` object as if it had never been serialized:

```
with open(state_path, 'rb') as f:
    state_after = pickle.load(f)
```

```
print(state_after.__dict__)
```

```
>>>
{'level': 1, 'lives': 3}
```

The problem with this approach is what happens as the game's features expand over time. Imagine that I want the player to earn points toward a high score. To track the player's points, I'd add a new field to the GameState class

```
class GameState:
    def __init__(self):
        self.level = 0
        self.lives = 4
        self.points = 0 # New field
```

Serializing the new version of the GameState class using pickle will work exactly as before. Here, I simulate the round-trip through a file by serializing to a string with dumps and back to an object with loads:

[Click here to view code image](#)

```
state = GameState()
serialized = pickle.dumps(state)
state_after = pickle.loads(serialized)
print(state_after.__dict__)
```

```
>>>
{'level': 0, 'lives': 4, 'points': 0}
```

But what happens to older saved GameState objects that the user may want to resume? Here, I unpickle an old game file by using a program with the new definition of the GameState class:

```
with open(state_path, 'rb') as f:
    state_after = pickle.load(f)

print(state_after.__dict__)
```

```
>>>
{'level': 1, 'lives': 3}
```

The points attribute is missing! This is especially confusing because the returned object is an instance of the new GameState class:

[Click here to view code image](#)

```
assert isinstance(state_after, GameState)
```

This behavior is a byproduct of the way the `pickle` module works. Its primary use case is making object serialization easy. As soon as your use of `pickle` moves beyond trivial usage, the module's functionality starts to break down in surprising ways.

Fixing these problems is straightforward using the `copyreg` built-in module. The `copyreg` module lets you register the functions responsible for serializing and deserializing Python objects, allowing you to control the behavior of `pickle` and make it more reliable.

## Default Attribute Values

In the simplest case, you can use a constructor with default arguments (see [Item 23: “Provide Optional Behavior with Keyword Arguments”](#) for background) to ensure that `GameState` objects will always have all attributes after unpickling. Here, I redefine the constructor this way:

[Click here to view code image](#)

```
class GameState:
    def __init__(self, level=0, lives=4, points=0):
        self.level = level
        self.lives = lives
        self.points = points
```

To use this constructor for pickling, I define a helper function that takes a `GameState` object and turns it into a tuple of parameters for the `copyreg` module. The returned tuple contains the function to use for unpickling and the parameters to pass to the unpickling function:

[Click here to view code image](#)

```
def pickle_game_state(game_state):
    kwargs = game_state.__dict__
    return unpickle_game_state, (kwargs,)
```

Now, I need to define the `unpickle_game_state` helper. This function takes serialized data and parameters from `pickle_game_state` and returns the



corresponding GameState object. It's a tiny wrapper around the constructor:

```
def unpickle_game_state(kwargs):  
    return GameState(**kwargs)
```

Now, I register these functions with the copyreg built-in module:

[Click here to view code image](#)

```
import copyreg  
  
copyreg.pickle(GameState, pickle_game_state)
```

After registration, serializing and deserializing works as before:

[Click here to view code image](#)

```
state = GameState()  
state.points += 1000  
serialized = pickle.dumps(state)  
state_after = pickle.loads(serialized)  
print(state_after.__dict__)  
  
>>>  
{'level': 0, 'lives': 4, 'points': 1000}
```

With this registration done, now I'll change the definition of GameState again to give the player a count of magic spells to use. This change is similar to when I added the points field to GameState:

[Click here to view code image](#)

```
class GameState:  
    def __init__(self, level=0, lives=4, points=0, magic=5):  
        self.level = level  
        self.lives = lives  
        self.points = points  
        self.magic = magic # New field
```

But unlike before, deserializing an old GameState object will result in valid game data instead of missing attributes. This works because unpickle\_game\_state calls the GameState constructor directly instead of using the pickle module's default behavior of saving and restoring only the attributes that belong to an object. The GameState constructor's keyword arguments have default values that will be used for any parameters that are

missing. This causes old game state files to receive the default value for the new magic field when they are deserialized:

[Click here to view code image](#)

```
print('Before:', state.__dict__)
state_after = pickle.loads(serialized)
print('After: ', state_after.__dict__)

>>>
Before: {'level': 0, 'lives': 4, 'points': 1000}
After:  {'level': 0, 'lives': 4, 'points': 1000, 'magic': 5}
```

## Versioning Classes

Sometimes you need to make backward-incompatible changes to your Python objects by removing fields. Doing so prevents the default argument approach above from working.

For example, say I realize that a limited number of lives is a bad idea, and I want to remove the concept of lives from the game. Here, I redefine the GameState class to no longer have a lives field:

[Click here to view code image](#)

```
class GameState:
    def __init__(self, level=0, points=0, magic=5):
        self.level = level
        self.points = points
        self.magic = magic
```

The problem is that this breaks deserialization of old game data. All fields from the old data, even ones removed from the class, will be passed to the GameState constructor by the unpickle\_game\_state function:

[Click here to view code image](#)

```
pickle.loads(serialized)
>>>
Traceback ...
TypeError: __init__() got an unexpected keyword argument
➡ 'lives'
```

I can fix this by adding a version parameter to the functions supplied to copyreg. New serialized data will have a version of 2 specified when pickling a new GameState object:

```
def pickle_game_state(game_state):
    kwargs = game_state.__dict__
    kwargs['version'] = 2
    return unpickle_game_state, (kwargs,)
```

Old versions of the data will not have a version argument present, which means I can manipulate the arguments passed to the GameState constructor accordingly:

```
def unpickle_game_state(kwargs):
    version = kwargs.pop('version', 1)
    if version == 1:
        del kwargs['lives']
    return GameState(**kwargs)
```

Now, deserializing an old object works properly:

[Click here to view code image](#)

```
copyreg.pickle(GameState, pickle_game_state)
print('Before:', state.__dict__)
state_after = pickle.loads(serialized)
print('After: ', state_after.__dict__)

>>>
Before: {'level': 0, 'lives': 4, 'points': 1000}
After:  {'level': 0, 'points': 1000, 'magic': 5}
```

I can continue using this approach to handle changes between future versions of the same class. Any logic I need to adapt an old version of the class to a new version of the class can go in the unpickle\_game\_state function.

## Stable Import Paths

One other issue you may encounter with pickle is breakage from renaming a class. Often over the life cycle of a program, you'll refactor your code by renaming classes and moving them to other modules. Unfortunately, doing so breaks the pickle module unless you're careful.

Here, I rename the GameState class to BetterGameState and remove the old class from the program entirely:

[Click here to view code image](#)

```
class BetterGameState:
    def __init__(self, level=0, points=0, magic=5):
        self.level = level
        self.points = points
        self.magic = magic
```

Attempting to deserialize an old GameState object now fails because the class can't be found:

[Click here to view code image](#)

```
pickle.loads(serialized)

>>>
Traceback ...
AttributeError: Can't get attribute 'GameState' on <module
➔ '__main__' from 'my_code.py'>
```

The cause of this exception is that the import path of the serialized object's class is encoded in the pickled data:

[Click here to view code image](#)

```
print(serialized)

>>>
b'\x80\x04\x95A\x00\x00\x00\x00\x00\x00\x00\x8c\x08__main__
➔\x94\x8c\tGameState\x94\x93\x94)\x81\x94}\x94(\x8c\x05level
➔\x94K\x00\x8c\x06points\x94K\x00\x8c\x05magic\x94K\x05ub.'
```

The solution is to use copyreg again. I can specify a stable identifier for the function to use for unpickling an object. This allows me to transition pickled data to different classes with different names when it's deserialized. It gives me a level of indirection:

[Click here to view code image](#)

```
copyreg.pickle(BetterGameState, pickle_game_state)
```

After I use `copyreg`, you can see that the import path to `unpickle_game_state` is encoded in the serialized data instead of `BetterGameState`:

[Click here to view code image](#)

```
state = BetterGameState()
serialized = pickle.dumps(state)
print(serialized)

>>>
b'\x80\x04\x95W\x00\x00\x00\x00\x00\x00\x00\x00\x8c\x08__main__
➔\x94\x8c\x13unpickle_game_state\x94\x93\x94}\x94(\x8c
➔\x05level\x94K\x00\x8c\x06points\x94K\x00\x8c\x05magic\x94K
➔\x05\x8c\x07version\x94K\x02u\x85\x94R\x94.'
```

The only gotcha is that I can't change the path of the module in which the `unpickle_game_state` function is present. Once I serialize data with a function, it must remain available on that import path for deserialization in the future.

## Things to Remember

- ◆ The `pickle` built-in module is useful only for serializing and deserializing objects between trusted programs.
- ◆ Deserializing previously pickled objects may break if the classes involved have changed over time (e.g., attributes have been added or removed).
- ◆ Use the `copyreg` built-in module with `pickle` to ensure backward compatibility for serialized objects.

## Item 69: Use `decimal` When Precision Is Paramount

Python is an excellent language for writing code that interacts with numerical data. Python's integer type can represent values of any practical size. Its double-precision floating point type complies with the IEEE 754 standard. The language also provides a standard complex number type for imaginary values. However, these aren't enough for every situation.

For example, say that I want to compute the amount to charge a customer for an international phone call. I know the time in minutes and seconds that the customer was on the phone (say, 3 minutes 42 seconds). I also have a set rate for the cost of calling Antarctica from the United States (\$1.45/minute). What should the charge be?

With floating point math, the computed charge seems reasonable

```
rate = 1.45
seconds = 3*60 + 42
cost = rate * seconds / 60
print(cost)
```

```
>>>
5.364999999999999
```

The result is 0.0001 short of the correct value (5.365) due to how IEEE 754 floating point numbers are represented. I might want to round up this value to 5.37 to properly cover all costs incurred by the customer. However, due to floating point error, rounding to the nearest whole cent actually reduces the final charge (from 5.364 to 5.36) instead of increasing it (from 5.365 to 5.37):

```
print(round(cost, 2))
>>>
5.36
```

The solution is to use the `Decimal` class from the `decimal` built-in module. The `Decimal` class provides fixed point math of 28 decimal places by default. It can go even higher, if required. This works around the precision issues in IEEE 754 floating point numbers. The class also gives you more control over rounding behaviors.

For example, redoing the Antarctica calculation with `Decimal` results in the exact expected charge instead of an approximation:

```
from decimal import Decimal

rate = Decimal('1.45')
seconds = Decimal(3*60 + 42)
cost = rate * seconds / Decimal(60)
print(cost)
```

```
>>>
5.365
```

Decimal instances can be given starting values in two different ways. The first way is by passing a `str` containing the number to the `Decimal` constructor. This ensures that there is no loss of precision due to the inherent nature of Python floating point numbers. The second way is by directly passing a `float` or an `int` instance to the constructor. Here, you can see that the two construction methods result in different behavior.

[Click here to view code image](#)

```
print(Decimal('1.45'))
print(Decimal(1.45))
```

```
>>>
1.45
1.4499999999999999555910790149937383830547332763671875
```

The same problem doesn't happen if I supply integers to the `Decimal` constructor:

```
print('456')
print(456)
```

```
>>>
456
456
```

If you care about exact answers, err on the side of caution and use the `str` constructor for the `Decimal` type.

Getting back to the phone call example, say that I also want to support very short phone calls between places that are much cheaper to connect (like Toledo and Detroit). Here, I compute the charge for a phone call that was 5 seconds long with a rate of \$0.05/minute:

[Click here to view code image](#)

```
rate = Decimal('0.05')
seconds = Decimal('5')
small_cost = rate * seconds / Decimal(60)
print(small_cost)
```

```
>>>
0.00416666666666666666666666666667
```

The result is so low that it is decreased to zero when I try to round it to the nearest whole cent. This won't do!

```
print(round(small_cost, 2))
```

```
>>>
0.00
```

Luckily, the `Decimal` class has a built-in function for rounding to exactly the decimal place needed with the desired rounding behavior. This works for the higher cost case from earlier:

[Click here to view code image](#)

```
from decimal import ROUND_UP

rounded = cost.quantize(Decimal('0.01'), rounding=ROUND_UP)
print(f'Rounded {cost} to {rounded}')
```

```
>>>
Rounded 5.365 to 5.37
```

Using the `quantize` method this way also properly handles the small usage case for short, cheap phone calls:.

[Click here to view code image](#)

```
rounded = small_cost.quantize(Decimal('0.01'),
                              rounding=ROUND_UP)
print(f'Rounded {small_cost} to {rounded}')
```

```
>>>
Rounded 0.00416666666666666666666666666667 to 0.01
```

While `Decimal` works great for fixed point numbers, it still has limitations in its precision (e.g.,  $1/3$  will be an approximation). For representing rational numbers with no limit to precision, consider using the `Fraction` class from the `fractions` built-in module.

## Things to Remember



- ✦ Python has built-in types and classes in modules that can represent practically every type of numerical value.
- ✦ The `Decimal` class is ideal for situations that require high precision and control over rounding behavior, such as computations of monetary values.
- ✦ Pass `str` instances to the `Decimal` constructor instead of `float` instances if it's important to compute exact answers and not floating point approximations.

## Item 70: Profile Before Optimizing

The dynamic nature of Python causes surprising behaviors in its runtime performance. Operations you might assume would be slow are actually very fast (e.g., string manipulation, generators). Language features you might assume would be fast are actually very slow (e.g., attribute accesses, function calls). The true source of slowdowns in a Python program can be obscure.

The best approach is to ignore your intuition and directly measure the performance of a program before you try to optimize it. Python provides a built-in *profiler* for determining which parts of a program are responsible for its execution time. This means you can focus your optimization efforts on the biggest sources of trouble and ignore parts of the program that don't impact speed (i.e., follow Amdahl's law).

For example, say that I want to determine why an algorithm in a program is slow. Here, I define a function that sorts a `list` of data using an insertion sort:

```
def insertion_sort(data):  
    result = []  
    for value in data:  
        insert_value(result, value)  
    return result
```

The core mechanism of the insertion sort is the function that finds the insertion point for each piece of data. Here, I define an extremely inefficient

version of the `insert_value` function that does a linear scan over the input array:

```
def insert_value(array, value):
    for i, existing in enumerate(array):
        if existing > value:
            array.insert(i, value)
    return
array.append(value)
```

To profile `insertion_sort` and `insert_value`, I create a data set of random numbers and define a test function to pass to the profiler:

[Click here to view code image](#)

```
from random import randint

max_size = 10**4
data = [randint(0, max_size) for _ in range(max_size)]
test = lambda: insertion_sort(data)
```

Python provides two built-in profilers: one that is pure Python (`profile`) and another that is a C-extension module (`cProfile`). The `cProfile` built-in module is better because of its minimal impact on the performance of your program while it's being profiled. The pure-Python alternative imposes a high overhead that skews the results.

## Note

When profiling a Python program, be sure that what you're measuring is the code itself and not external systems. Beware of functions that access the network or resources on disk. These may appear to have a large impact on your program's execution time because of the slowness of the underlying systems. If your program uses a cache to mask the latency of slow resources like these, you should ensure that it's properly warmed up before you start profiling.

Here, I instantiate a `Profile` object from the `cProfile` module and run the test function through it using the `runcall` method:

```
from cProfile import Profile
```

```
profiler = Profile()
profiler.runcall(test)
```

When the test function has finished running, I can extract statistics about its performance by using the `pstats` built-in module and its `Stats` class.

Various methods on a `stats` object adjust how to select and sort the profiling information to show only the things I care about:

```
from pstats import Stats

stats = Stats(profiler)
stats.strip_dirs()
stats.sort_stats('cumulative')
stats.print_stats()
```

The output is a table of information organized by function. The data sample is taken only from the time the profiler was active, during the `runcall` method above:

[Click here to view code image](#)

```
>>>
      20003 function calls in 1.320 seconds

      Ordered by: cumulative time

      ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
          1    0.000    0.000    1.320    1.320 main.py:35(<lambda>)
          1    0.003    0.003    1.320    1.320
main.py:10(insertion_sort)
      10000    1.306    0.000    1.317    0.000
main.py:20(insert_value)
       9992    0.011    0.000    0.011    0.000 {method 'insert' of
'list' objects}
          8    0.000    0.000    0.000    0.000 {method 'append' of
'list' objects}
```

Here's a quick guide to what the profiler statistics columns mean:

- `ncalls`: The number of calls to the function during the profiling period.
- `tottime`: The number of seconds spent executing the function, excluding time spent executing other functions it calls.

- `tottime percall`: The average number of seconds spent in the function each time it is called, excluding time spent executing other functions it calls. This is `tottime` divided by `ncalls`.
- `cumtime`: The cumulative number of seconds spent executing the function, including time spent in all other functions it calls.
- `cumtime percall`: The average number of seconds spent in the function each time it is called, including time spent in all other functions it calls. This is `cumtime` divided by `ncalls`.

Looking at the profiler statistics table above, I can see that the biggest use of CPU in my test is the cumulative time spent in the `insert_value` function. Here, I redefine that function to use the `bisect` built-in module (see [Item 72: “Consider Searching Sorted Sequences with `bisect`”](#)):

```
from bisect import bisect_left

def insert_value(array, value):
    i = bisect_left(array, value)
    array.insert(i, value)
```

I can run the profiler again and generate a new table of profiler statistics. The new function is much faster, with a cumulative time spent that is nearly 100 times smaller than with the previous `insert_value` function:

[Click here to view code image](#)

```
>>>
    30003 function calls in 0.017 seconds

    Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
         1    0.000    0.000    0.017    0.017 main.py:35(<lambda>)
         1    0.002    0.002    0.017    0.017
main.py:10(insertion_sort)
    10000    0.003    0.000    0.015    0.000
main.py:110(insert_value)
    10000    0.008    0.000    0.008    0.000 {method 'insert' of
'list' objects}
    10000    0.004    0.000    0.004    0.000 {built-in method
_bisect.bisect_left}
```

Sometimes when you're profiling an entire program, you might find that a common utility function is responsible for the majority of execution time. The default output from the profiler makes such a situation difficult to understand because it doesn't show that the utility function is called by many different parts of your program.

For example, here the `my_utility` function is called repeatedly by two different functions in the program:

```
def my_utility(a, b):
    c = 1
    for i in range(100):
        c += a * b

def first_func():
    for _ in range(1000):
        my_utility(4, 5)

def second_func():
    for _ in range(10):
        my_utility(1, 3)

def my_program():
    for _ in range(20):
        first_func()
        second_func()
```

Profiling this code and using the default `print_stats` output generates statistics that are confusing:

[Click here to view code image](#)

```
>>>
```

```
20242 function calls in 0.118 seconds
```

```
Ordered by: cumulative time
```

	ncalls	totttime	percall	cumtime	percall
filename:lineno(function)					
	1	0.000	0.000	0.118	0.118
main.py:176(my_program)					
	20	0.003	0.000	0.117	0.006
main.py:168(first_func)					
	20200	0.115	0.000	0.115	0.000
main.py:161(my_utility)					

```

        20      0.000      0.000      0.001      0.000
main.py:172(second_func)

```

The `my_utility` function is clearly the source of most execution time, but it's not immediately obvious why that function is called so much. If you search through the program's code, you'll find multiple call sites for `my_utility` and still be confused.

To deal with this, the Python profiler provides the `print_callers` method to show which callers contributed to the profiling information of each function:

```
stats.print_callers()
```

This profiler statistics table shows functions called on the left and which function was responsible for making the call on the right. Here, it's clear that `my_utility` is most used by `first_func`:

[Click here to view code image](#)

```
>>>
```

```
Ordered by: cumulative time
```

Function	was called by...	ncalls	totttime
cumtime			
main.py:176(my_program)	<-		
main.py:168(first_func)	<-	20	0.003
0.117 main.py:176(my_program)			
main.py:161(my_utility)	<-	20000	0.114
0.114 main.py:168(first_func)			
		200	0.001
0.001 main.py:172(second_func)			
Profiling.md:172(second_func)	<-	20	0.000
0.001 main.py:176(my_program)			

## Things to Remember

- ✦ It's important to profile Python programs before optimizing because the sources of slowdowns are often obscure.
- ✦ Use the `cProfile` module instead of the `profile` module because it provides more accurate profiling information.

- ♦ The `Profile` object's `runcall` method provides everything you need to profile a tree of function calls in isolation.
- ♦ The `stats` object lets you select and print the subset of profiling information you need to see to understand your program's performance.

## Item 71: Prefer `deque` for Producer–Consumer Queues

A common need in writing programs is a first-in, first-out (FIFO) queue, which is also known as a producer–consumer queue. A FIFO queue is used when one function gathers values to process and another function handles them in the order in which they were received. Often, programmers use Python's built-in `list` type as a FIFO queue.

For example, say that I have a program that's processing incoming emails for long-term archival, and it's using a `list` for a producer–consumer queue. Here, I define a class to represent the messages:

[Click here to view code image](#)

```
class Email:
    def __init__(self, sender, receiver, message):
        self.sender = sender
        self.receiver = receiver
        self.message = message
    ...
```

I also define a placeholder function for receiving a single email, presumably from a socket, the file system, or some other type of I/O system. The implementation of this function doesn't matter; what's important is its interface: It will either return an `Email` instance or raise a `NoEmailError` exception:

[Click here to view code image](#)

```
class NoEmailError(Exception):
    pass

def try_receive_email():
    # Returns an Email instance or raises NoEmailError
    ...
```

The producing function receives emails and enqueues them to be consumed at a later time. This function uses the `append` method on the `list` to add new messages to the end of the queue so they are processed after all messages that were previously received:

```
def produce_emails(queue):
    while True:
        try:
            email = try_receive_email()
        except NoEmailError:
            return
        else:
            queue.append(email)  # Producer
```

The consuming function does something useful with the emails. This function calls `pop(0)` on the queue, which removes the very first item from the `list` and returns it to the caller. By always processing items from the beginning of the queue, the consumer ensures that the items are processed in the order in which they were received:

[Click here to view code image](#)

```
def consume_one_email(queue):
    if not queue:
        return
    email = queue.pop(0)  # Consumer
    # Index the message for long-term archival
    ...
```

Finally, I need a looping function that connects the pieces together. This function alternates between producing and consuming until the `keep_running` function returns `False` (see [Item 60: “Achieve Highly Concurrent I/O with Coroutines”](#) on how to do this concurrently):

```
def loop(queue, keep_running):
    while keep_running():
        produce_emails(queue)
        consume_one_email(queue)

def my_end_func():
    ...

loop([], my_end_func)
```



Why not process each Email message in `produce_emails` as it's returned by `try_receive_email`? It comes down to the trade-off between latency and throughput. When using producer–consumer queues, you often want to minimize the latency of accepting new items so they can be collected as fast as possible. The consumer can then process through the backlog of items at a consistent pace—one item per loop in this case—which provides a stable performance profile and consistent throughput at the cost of end-to-end latency (see [Item 55: “Use queue to Coordinate Work Between Threads”](#) for related best practices).

Using a `list` for a producer–consumer queue like this works fine up to a point, but as the *cardinality*—the number of items in the list—increases, the `list` type's performance can degrade superlinearly. To analyze the performance of using `list` as a FIFO queue, I can run some micro-benchmarks using the `timeit` built-in module. Here, I define a benchmark for the performance of adding new items to the queue using the `append` method of `list` (matching the producer function's usage):

[Click here to view code image](#)

```
import timeit

def print_results(count, tests):
    avg_iteration = sum(tests) / len(tests)
    print(f'Count {count:>5,} takes {avg_iteration:.6f}s')
    return count, avg_iteration

def list_append_benchmark(count):
    def run(queue):
        for i in range(count):
            queue.append(i)

    tests = timeit.repeat(
        setup='queue = []',
        stmt='run(queue)',
        globals=locals(),
        repeat=1000,
        number=1)

    return print_results(count, tests)
```

Running this benchmark function with different levels of cardinality lets me compare its performance in relationship to data size:

[Click here to view code image](#)

```
def print_delta(before, after):
    before_count, before_time = before
    after_count, after_time = after
    growth = 1 + (after_count - before_count) / before_count
    slowdown = 1 + (after_time - before_time) / before_time
    print(f'{growth:>4.1f}x data size, {slowdown:>4.1f}x time')
```

```
baseline = list_append_benchmark(500)
for count in (1_000, 2_000, 3_000, 4_000, 5_000):
    comparison = list_append_benchmark(count)
    print_delta(baseline, comparison)
```

```
>>>
```

```
Count    500 takes 0.000039s
```

```
Count 1,000 takes 0.000073s
    2.0x data size,  1.9x time
```

```
Count 2,000 takes 0.000121s
    4.0x data size,  3.1x time
```

```
Count 3,000 takes 0.000172s
    6.0x data size,  4.5x time
```

```
Count 4,000 takes 0.000240s
    8.0x data size,  6.2x time
```

```
Count 5,000 takes 0.000304s
   10.0x data size,  7.9x time
```

This shows that the append method takes roughly constant time for the `list` type, and the total time for enqueueing scales linearly as the data size increases. There is overhead for the `list` type to increase its capacity under the covers as new items are added, but it's reasonably low and is amortized across repeated calls to `append`.

Here, I define a similar benchmark for the `pop(0)` call that removes items from the beginning of the queue (matching the consumer function's usage):

```

def list_pop_benchmark(count):
    def prepare():
        return list(range(count))

    def run(queue):
        while queue:
            queue.pop(0)

    tests = timeit.repeat(
        setup='queue = prepare()',
        stmt='run(queue)',
        globals=locals(),
        repeat=1000,
        number=1)

    return print_results(count, tests)

```

I can similarly run this benchmark for queues of different sizes to see how performance is affected by cardinality:

[Click here to view code image](#)

```

baseline = list_pop_benchmark(500)
for count in (1_000, 2_000, 3_000, 4_000, 5_000):
    comparison = list_pop_benchmark(count)
    print_delta(baseline, comparison)

```

```

>>>
Count    500 takes 0.000050s

Count 1,000 takes 0.000133s
      2.0x data size,  2.7x time

Count 2,000 takes 0.000347s
      4.0x data size,  6.9x time

Count 3,000 takes 0.000663s
      6.0x data size, 13.2x time

Count 4,000 takes 0.000943s
      8.0x data size, 18.8x time

Count 5,000 takes 0.001481s
     10.0x data size, 29.5x time

```

Surprisingly, this shows that the total time for dequeuing items from a list with `pop(0)` scales quadratically as the length of the queue increases. The

cause is that `pop(0)` needs to move every item in the `list` back an index, effectively reassigning the entire list's contents. I need to call `pop(0)` for every item in the `list`, and thus I end up doing roughly `len(queue) * len(queue)` operations to consume the queue. This doesn't scale.

Python provides the `deque` class from the `collections` built-in module to solve this problem. `deque` is a *double-ended queue* implementation. It provides constant time operations for inserting or removing items from its beginning or end. This makes it ideal for FIFO queues.

To use the `deque` class, the call to `append` in `produce_emails` can stay the same as it was when using a `list` for the queue. The `list.pop` method call in `consume_one_email` must change to call the `deque.popleft` method with no arguments instead. And the `loop` method must be called with a `deque` instance instead of a `list`. Everything else stays the same. Here, I redefine the one function affected to use the new method and run `loop` again:

```
import collections

def consume_one_email(queue):
    if not queue:
        return
    email = queue.popleft() # Consumer
    # Process the email message
    ...

def my_end_func():
    ...

loop(collections.deque(), my_end_func)
```

I can run another version of the benchmark to verify that `append` performance (matching the producer function's usage) has stayed roughly the same (modulo a constant factor):

[Click here to view code image](#)

```
def deque_append_benchmark(count):
    def prepare():
        return collections.deque()
    def run(queue):
        for i in range(count):
            queue.append(i)
```

```

tests = timeit.repeat(
    setup='queue = prepare()',
    stmt='run(queue)',
    globals=locals(),
    repeat=1000,
    number=1)
return print_results(count, tests)

baseline = deque_append_benchmark(500)
for count in (1_000, 2_000, 3_000, 4_000, 5_000):
    comparison = deque_append_benchmark(count)
    print_delta(baseline, comparison)

```

>>>

Count 500 takes 0.000029s

Count 1,000 takes 0.000059s  
2.0x data size, 2.1x time

Count 2,000 takes 0.000121s  
4.0x data size, 4.2x time

Count 3,000 takes 0.000171s  
6.0x data size, 6.0x time

Count 4,000 takes 0.000243s  
8.0x data size, 8.5x time

Count 5,000 takes 0.000295s  
10.0x data size, 10.3x time

And I can benchmark the performance of calling `popleft` to mimic the consumer function's usage of deque:

[Click here to view code image](#)

```

def dequeue_popleft_benchmark(count):
    def prepare():
        return collections.deque(range(count))

    def run(queue):
        while queue:
            queue.popleft()

    tests = timeit.repeat(
        setup='queue = prepare()',

```

```

        stmt='run(queue)',
        globals=locals(),
        repeat=1000,
        number=1)

    return print_results(count, tests)

baseline = dequeue_popleft_benchmark(500)
for count in (1_000, 2_000, 3_000, 4_000, 5_000):
    comparison = dequeue_popleft_benchmark(count)
    print_delta(baseline, comparison)

>>>
Count    500 takes 0.000024s

Count 1,000 takes 0.000050s
      2.0x data size,  2.1x time

Count 2,000 takes 0.000100s
      4.0x data size,  4.2x time

Count 3,000 takes 0.000152s
      6.0x data size,  6.3x time

Count 4,000 takes 0.000207s
      8.0x data size,  8.6x time

Count 5,000 takes 0.000265s
     10.0x data size, 11.0x time

```

The `popleft` usage scales linearly instead of displaying the superlinear behavior of `pop(0)` that I measured before—hooray! If you know that the performance of a program critically depends on the speed of producer–consumer queues, then `deque` is a great choice. If you’re not sure, then you should instrument your program to find out (see [Item 70: “Profile Before Optimizing”](#)).

## Things to Remember

- ◆ The `list` type can be used as a FIFO queue by having the producer call `append` to add items and the consumer call `pop(0)` to receive items. However, this may cause problems because the performance of `pop(0)` degrades superlinearly as the queue length increases.

- ♦ The deque class from the collections built-in module takes constant time—regardless of length—for append and popleft, making it ideal for FIFO queues.

## Item 72: Consider Searching Sorted Sequences with `bisect`

It's common to find yourself with a large amount of data in memory as a sorted list that you then want to search. For example, you may have loaded an English language dictionary to use for spell checking, or perhaps a list of dated financial transactions to audit for correctness.

Regardless of the data your specific program needs to process, searching for a specific value in a list takes linear time proportional to the list's length when you call the `index` method:

```
data = list(range(10**5))
index = data.index(91234)
assert index == 91234
```

If you're not sure whether the exact value you're searching for is in the list, then you may want to search for the closest index that is equal to or exceeds your goal value. The simplest way to do this is to linearly scan the list and compare each item to your goal value:

[Click here to view code image](#)

```
def find_closest(sequence, goal):
    for index, value in enumerate(sequence):
        if goal < value:
            return index
    raise ValueError(f'{goal} is out of bounds')

index = find_closest(data, 91234.56)
assert index == 91235
```

Python's built-in `bisect` module provides better ways to accomplish these types of searches through ordered lists. You can use the `bisect_left` function to do an efficient binary search through any sequence of sorted items. The index it returns will either be where the item is already present in

the list or where you'd want to insert the item in the list to keep it in sorted order:

[Click here to view code image](#)

```
from bisect import bisect_left

index = bisect_left(data, 91234) # Exact match
assert index == 91234

index = bisect_left(data, 91234.56) # Closest match
assert index == 91235
```

The complexity of the binary search algorithm used by the bisect module is logarithmic. This means searching in a list of length 1 million takes roughly the same amount of time with bisect as linearly searching a list of length 20 using the list.index method ( $\text{math.log2}(10^6) \approx 19.93\dots$ ). It's way faster!

I can verify this speed improvement for the example from above by using the timeit built-in module to run a micro-benchmark:

[Click here to view code image](#)

```
import random
import timeit

size = 10**5
iterations = 1000

data = list(range(size))
to_lookup = [random.randint(0, size)
              for _ in range(iterations)]

def run_linear(data, to_lookup):
    for index in to_lookup:
        data.index(index)

def run_bisect(data, to_lookup):
    for index in to_lookup:
        bisect_left(data, index)

baseline = timeit.timeit(
    stmt='run_linear(data, to_lookup)',
    globals=globals(),
```



```

    number=10)
print(f'Linear search takes {baseline:.6f}s')

comparison = timeit.timeit(
    stmt='run_bisect(data, to_lookup)',
    globals=globals(),
    number=10)
print(f'Bisect search takes {comparison:.6f}s')

slowdown = 1 + ((baseline - comparison) / comparison)
print(f'{slowdown:.1f}x time')
>>>
Linear search takes 5.370117s
Bisect search takes 0.005220s
1028.7x time

```

The best part about `bisect` is that it's not limited to the `list` type; you can use it with any Python object that acts like a sequence (see [Item 43: “Inherit from `collections.abc` for Custom Container Types](#)” for how to do that). The module also provides additional features for more advanced situations (see `help(bisect)`).

## Things to Remember

- ✦ Searching sorted data contained in a `list` takes linear time using the `index` method or a `for` loop with simple comparisons.
- ✦ The `bisect` built-in module's `bisect_left` function takes logarithmic time to search for values in sorted lists, which can be orders of magnitude faster than other approaches.

## Item 73: Know How to Use `heapq` for Priority Queues

One of the limitations of Python's other queue implementations (see [Item 71: “Prefer `deque` for Producer–Consumer Queues](#)” and [Item 55: “Use queue to Coordinate Work Between Threads](#)”) is that they are first-in, first-out (FIFO) queues: Their contents are sorted by the order in which they were received. Often, you need a program to process items in order of relative importance instead. To accomplish this, a *priority queue* is the right tool for the job.

For example, say that I'm writing a program to manage books borrowed from a library. There are people constantly borrowing new books. There are people returning their borrowed books on time. And there are people who need to be reminded to return their overdue books. Here, I define a class to represent a book that's been borrowed:

```
class Book:
    def __init__(self, title, due_date):
        self.title = title
        self.due_date = due_date
```

I need a system that will send reminder messages when each book passes its due date. Unfortunately, I can't use a FIFO queue for this because the amount of time each book is allowed to be borrowed varies based on its recency, popularity, and other factors. For example, a book that is borrowed today may be due back later than a book that's borrowed tomorrow. Here, I achieve this behavior by using a standard `list` and sorting it by `due_date` each time a new `Book` is added:

[Click here to view code image](#)

```
def add_book(queue, book):
    queue.append(book)
    queue.sort(key=lambda x: x.due_date, reverse=True)

queue = []
add_book(queue, Book('Don Quixote', '2019-06-07'))
add_book(queue, Book('Frankenstein', '2019-06-05'))
add_book(queue, Book('Les Misérables', '2019-06-08'))
add_book(queue, Book('War and Peace', '2019-06-03'))
```

If I can assume that the queue of borrowed books is always in sorted order, then all I need to do to check for overdue books is to inspect the final element in the `list`. Here, I define a function to return the next overdue book, if any, and remove it from the queue:

[Click here to view code image](#)

```
class NoOverdueBooks(Exception):
    pass

def next_overdue_book(queue, now):
    if queue:
```

```

        book = queue[-1]
        if book.due_date < now:
            queue.pop()
            return book

    raise NoOverdueBooks

```

I can call this function repeatedly to get overdue books to remind people about in the order of most overdue to least overdue:

```

now = '2019-06-10'

found = next_overdue_book(queue, now)
print(found.title)

found = next_overdue_book(queue, now)
print(found.title)

>>>
War and Peace
Frankenstein

```

If a book is returned before the due date, I can remove the scheduled reminder message by removing the Book from the list:

[Click here to view code image](#)

```

def return_book(queue, book):
    queue.remove(book)

queue = []
book = Book('Treasure Island', '2019-06-04')

add_book(queue, book)
print('Before return:', [x.title for x in queue])

return_book(queue, book)
print('After return: ', [x.title for x in queue])

>>>
Before return: ['Treasure Island']
After return: []

```

And I can confirm that when all books are returned, the `return_book` function will raise the right exception (see [Item 20: “Prefer Raising Exceptions to Returning None”](#)):

```

try:
    next_overdue_book(queue, now)
except NoOverdueBooks:
    pass          # Expected
else:
    assert False  # Doesn't happen

```

However, the computational complexity of this solution isn't ideal. Although checking for and removing an overdue book has a constant cost, every time I add a book, I pay the cost of sorting the whole `list` again. If I have `len(queue)` books to add, and the cost of sorting them is roughly `len(queue) * math.log(len(queue))`, the time it takes to add books will grow superlinearly (`len(queue) * len(queue) * math.log(len(queue))`).

Here, I define a micro-benchmark to measure this performance behavior experimentally by using the `timeit` built-in module (see [Item 71: “Prefer deque for Producer–Consumer Queues”](#) for the implementation of `print_results` and `print_delta`):

```

import random
import timeit

def print_results(count, tests):
    ...

def print_delta(before, after):
    ...

def list_overdue_benchmark(count):
    def prepare():
        to_add = list(range(count))
        random.shuffle(to_add)
        return [], to_add

    def run(queue, to_add):
        for i in to_add:
            queue.append(i)
            queue.sort(reverse=True)

        while queue:
            queue.pop()

    tests = timeit.repeat(
        setup='queue, to_add = prepare()',
        stmt=f'run(queue, to_add)',

```

```

        globals=locals(),
        repeat=100,
        number=1)

    return print_results(count, tests)

```

I can verify that the runtime of adding and removing books from the queue scales superlinearly as the number of books being borrowed increases:

[Click here to view code image](#)

```

baseline = list_overdue_benchmark(500)
for count in (1_000, 1_500, 2_000):
    comparison = list_overdue_benchmark(count)
    print_delta(baseline, comparison)

```

```

>>>
Count    500 takes 0.001138s

Count 1,000 takes 0.003317s
      2.0x data size,   2.9x time

Count 1,500 takes 0.007744s
      3.0x data size,   6.8x time

Count 2,000 takes 0.014739s
      4.0x data size,  13.0x time

```

When a book is returned before the due date, I need to do a linear scan in order to find the book in the queue and remove it. Removing a book causes all subsequent items in the list to be shifted back an index, which has a high cost that also scales superlinearly. Here, I define another micro-benchmark to test the performance of returning a book using this function:

```

def list_return_benchmark(count):
    def prepare():
        queue = list(range(count))
        random.shuffle(queue)

        to_return = list(range(count))
        random.shuffle(to_return)

    return queue, to_return

def run(queue, to_return):
    for i in to_return:

```

```

        queue.remove(i)

tests = timeit.repeat(
    setup='queue, to_return = prepare()',
    stmt=f'run(queue, to_return)',
    globals=locals(),
    repeat=100,
    number=1)

return print_results(count, tests)

```

And again, I can verify that indeed the performance degrades superlinearly as the number of books increases:

[Click here to view code image](#)

```

baseline = list_return_benchmark(500)
for count in (1_000, 1_500, 2_000):
    comparison = list_return_benchmark(count)
    print_delta(baseline, comparison)

```

```

>>>
Count    500 takes 0.000898s

Count 1,000 takes 0.003331s
      2.0x data size,   3.7x time

Count 1,500 takes 0.007674s
      3.0x data size,   8.5x time

Count 2,000 takes 0.013721s
      4.0x data size,  15.3x time

```

Using the methods of `list` may work for a tiny library, but it certainly won't scale to the size of the Great Library of Alexandria, as I want it to!

Fortunately, Python has the built-in `heapq` module that solves this problem by implementing priority queues efficiently. A *heap* is a data structure that allows for a `list` of items to be maintained where the computational complexity of adding a new item or removing the smallest item has logarithmic computational complexity (i.e., even better than linear scaling). In this library example, smallest means the book with the earliest due date. The best part about this module is that you don't have to understand how heaps are implemented in order to use its functions correctly.

Here, I reimplement the `add_book` function using the `heapq` module. The queue is still a plain `list`. The `heappush` function replaces the `list.append` call from before. And I no longer have to call `list.sort` on the queue:

```
from heapq import heappush

def add_book(queue, book):
    heappush(queue, book)
```

If I try to use this with the `Book` class as previously defined, I get this somewhat cryptic error:

[Click here to view code image](#)

```
queue = []
add_book(queue, Book('Little Women', '2019-06-05'))
add_book(queue, Book('The Time Machine', '2019-05-30'))

>>>
Traceback ...
TypeError: '<' not supported between instances of 'Book' and
➔ 'Book'
```

The `heapq` module requires items in the priority queue to be comparable and have a natural sort order (see [Item 14: “Sort by Complex Criteria Using the `key` Parameter](#)” for details). You can quickly give the `Book` class this behavior by using the `total_ordering` class decorator from the `functools` built-in module (see [Item 51: “Prefer Class Decorators Over Metaclasses for Composable Class Extensions](#)” for background) and implementing the `__lt__` special method (see [Item 43: “Inherit from `collections.abc` for Custom Container Types](#)” for background). Here, I redefine the class with a less-than method that simply compares the `due_date` fields between two `Book` instances:

```
import functools

@functools.total_ordering
class Book:
    def __init__(self, title, due_date):
        self.title = title
        self.due_date = due_date
```

```
def __lt__(self, other):  
    return self.due_date < other.due_date
```

Now, I can add books to the priority queue by using the `heapq.heappush` function without issues:

[Click here to view code image](#)

```
queue = []  
add_book(queue, Book('Pride and Prejudice', '2019-06-01'))  
add_book(queue, Book('The Time Machine', '2019-05-30'))  
add_book(queue, Book('Crime and Punishment', '2019-06-06'))  
add_book(queue, Book('Wuthering Heights', '2019-06-12'))
```

Alternatively, I can create a `list` with all of the books in any order and then use the `sort` method of `list` to produce the heap:

[Click here to view code image](#)

```
queue = [  
    Book('Pride and Prejudice', '2019-06-01'),  
    Book('The Time Machine', '2019-05-30'),  
    Book('Crime and Punishment', '2019-06-06'),  
    Book('Wuthering Heights', '2019-06-12'),  
]  
queue.sort()
```

Or I can use the `heapq.heapify` function to create a heap in linear time (as opposed to the `sort` method's  $\text{len}(\text{queue}) * \log(\text{len}(\text{queue}))$  complexity):

[Click here to view code image](#)

```
from heapq import heapify  
  
queue = [  
    Book('Pride and Prejudice', '2019-06-01'),  
    Book('The Time Machine', '2019-05-30'),  
    Book('Crime and Punishment', '2019-06-06'),  
    Book('Wuthering Heights', '2019-06-12'),  
]  
heapify(queue)
```

To check for overdue books, I inspect the first element in the `list` instead of the last, and then I use the `heapq.heappop` function instead of the `list.pop` function:



[Click here to view code image](#)

```
from heapq import heappop

def next_overdue_book(queue, now):
    if queue:
        book = queue[0]                # Most overdue first
        if book.due_date < now:
            heappop(queue)             # Remove the overdue book
            return book

    raise NoOverdueBooks
```

Now, I can find and remove overdue books in order until there are none left for the current time:

[Click here to view code image](#)

```
now = '2019-06-02'

book = next_overdue_book(queue, now)
print(book.title)

book = next_overdue_book(queue, now)
print(book.title)

try:
    next_overdue_book(queue, now)
except NoOverdueBooks:
    pass                                # Expected
else:
    assert False                       # Doesn't happen

>>>
The Time Machine
Pride and Prejudice
```

I can write another micro-benchmark to test the performance of this implementation that uses the `heapq` module:

```
def heap_overdue_benchmark(count):
    def prepare():
        to_add = list(range(count))
        random.shuffle(to_add)
        return [], to_add

    def run(queue, to_add):
```

```

    for i in to_add:
        heappush(queue, i)
    while queue:
        heappop(queue)

tests = timeit.repeat(
    setup='queue, to_add = prepare()',
    stmt=f'run(queue, to_add)',
    globals=locals(),
    repeat=100,
    number=1)

return print_results(count, tests)

```

This benchmark experimentally verifies that the heap-based priority queue implementation scales much better (roughly  $\log(\text{len}(\text{queue}))$  \*  $\text{len}(\text{queue})$ ), without superlinearly degrading performance:

[Click here to view code image](#)

```

baseline = heap_overdue_benchmark(500)
for count in (1_000, 1_500, 2_000):
    comparison = heap_overdue_benchmark(count)
    print_delta(baseline, comparison)

```

```

>>>
Count    500 takes 0.000150s

Count 1,000 takes 0.000325s
    2.0x data size,   2.2x time

Count 1,500 takes 0.000528s
    3.0x data size,   3.5x time

Count 2,000 takes 0.000658s
    4.0x data size,   4.4x time

```

With the `heapq` implementation, one question remains: How should I handle returns that are on time? The solution is to never remove a book from the priority queue until its due date. At that time, it will be the first item in the list, and I can simply ignore the book if it's already been returned. Here, I implement this behavior by adding a new field to track the book's return status:

```
@functools.total_ordering
class Book:
    def __init__(self, title, due_date):
        self.title = title
        self.due_date = due_date
        self.returned = False # New field
    ...
```

Then, I change the `next_overdue_book` function to repeatedly ignore any book that's already been returned:

```
def next_overdue_book(queue, now):
    while queue:
        book = queue[0]
        if book.returned:
            heappop(queue)
            continue

        if book.due_date < now:
            heappop(queue)
            return book

    break

    raise NoOverdueBooks
```

This approach makes the `return_book` function extremely fast because it makes no modifications to the priority queue:

```
def return_book(queue, book):
    book.returned = True
```

The downside of this solution for returns is that the priority queue may grow to the maximum size it would have needed if all books from the library were checked out and went overdue. Although the queue operations will be fast thanks to `heapq`, this storage overhead may take significant memory (see [Item 81: “Use `tracemalloc` to Understand Memory Usage and Leaks](#)” for how to debug such usage).

That said, if you're trying to build a robust system, you need to plan for the worst-case scenario; thus, you should expect that it's possible for every library book to go overdue for some reason (e.g., a natural disaster closes the road to the library). This memory cost is a design consideration that you should have already planned for and mitigated through additional

constraints (e.g., imposing a maximum number of simultaneously lent books).

Beyond the priority queue primitives that I’ve used in this example, the `heapq` module provides additional functionality for advanced use cases (see `help(heapq)`). The module is a great choice when its functionality matches the problem you’re facing (see the `queue.PriorityQueue` class for another thread-safe option).

## Things to Remember

- ◆ Priority queues allow you to process items in order of importance instead of in first-in, first-out order.
- ◆ If you try to use `list` operations to implement a priority queue, your program’s performance will degrade superlinearly as the queue grows.
- ◆ The `heapq` built-in module provides all of the functions you need to implement a priority queue that scales efficiently.
- ◆ To use `heapq`, the items being prioritized must have a natural sort order, which requires special methods like `__lt__` to be defined for classes.

## Item 74: Consider `memoryview` and `bytearray` for Zero-Copy Interactions with `bytes`

Although Python isn’t able to parallelize CPU-bound computation without extra effort (see [Item 64: “Consider `concurrent.futures` for True Parallelism](#)”), it is able to support high-throughput, parallel I/O in a variety of ways (see [Item 53: “Use Threads for Blocking I/O, Avoid for Parallelism](#)” and [Item 60: “Achieve Highly Concurrent I/O with Coroutines](#)”). That said, it’s surprisingly easy to use these I/O tools the wrong way and reach the conclusion that the language is too slow for even I/O-bound workloads.

For example, say that I’m building a media server to stream television or movies over a network to users so they can watch without having to download the video data in advance. One of the key features of such a

system is the ability for users to move forward or backward in the video playback so they can skip or repeat parts. In the client program, I can implement this by requesting a chunk of data from the server corresponding to the new time index selected by the user:

[Click here to view code image](#)

```
def timecode_to_index(video_id, timecode):
    ...
    # Returns the byte offset in the video data

def request_chunk(video_id, byte_offset, size):
    ...
    # Returns size bytes of video_id's data from the offset

video_id = ...
timecode = '01:09:14:28'
byte_offset = timecode_to_index(video_id, timecode)
size = 20 * 1024 * 1024
video_data = request_chunk(video_id, byte_offset, size)
```

How would you implement the server-side handler that receives the `request_chunk` request and returns the corresponding 20 MB chunk of video data? For the sake of this example, I assume that the command and control parts of the server have already been hooked up (see [Item 61: “Know How to Port Threaded I/O to `asyncio`”](#) for what that requires). I focus here on the last steps where the requested chunk is extracted from gigabytes of video data that’s cached in memory and is then sent over a socket back to the client. Here’s what the implementation would look like:

[Click here to view code image](#)

```
socket = ...                # socket connection to client
video_data = ...             # bytes containing data for video_id
byte_offset = ...            # Requested starting position
size = 20 * 1024 * 1024      # Requested chunk size

chunk = video_data[byte_offset:byte_offset + size]
socket.send(chunk)
```

The latency and throughput of this code will come down to two factors: how much time it takes to slice the 20 MB video chunk from `video_data`, and how much time the socket takes to transmit that data to the client. If I

assume that the socket is infinitely fast, I can run a micro-benchmark by using the `timeit` built-in module to understand the performance characteristics of slicing `bytes` instances this way to create chunks (see [Item 11: “Know How to Slice Sequences”](#) for background):

[Click here to view code image](#)

```
import timeit

def run_test():
    chunk = video_data[byte_offset:byte_offset + size]
    # Call socket.send(chunk), but ignoring for benchmark

result = timeit.timeit(
    stmt='run_test()',
    globals=globals(),
    number=100) / 100

print(f'{result:0.9f} seconds')

>>>
0.004925669 seconds
```

It took roughly 5 milliseconds to extract the 20 MB slice of data to transmit to the client. That means the overall throughput of my server is limited to a theoretical maximum of 20 MB / 5 milliseconds = 7.3 GB / second, since that's the fastest I can extract the video data from memory. My server will also be limited to 1 CPU-second / 5 milliseconds = 200 clients requesting new chunks in parallel, which is tiny compared to the tens of thousands of simultaneous connections that tools like the `asyncio` built-in module can support. The problem is that slicing a `bytes` instance causes the underlying data to be copied, which takes CPU time.

A better way to write this code is by using Python's built-in `memoryview` type, which exposes CPython's high-performance *buffer protocol* to programs. The buffer protocol is a low-level C API that allows the Python runtime and C extensions to access the underlying data buffers that are behind objects like `bytes` instances. The best part about `memoryview` instances is that slicing them results in another `memoryview` instance without copying the underlying data. Here, I create a `memoryview` wrapping a `bytes` instance and inspect a slice of it:

[Click here to view code image](#)

```
data = b'shave and a haircut, two bits'
view = memoryview(data)
chunk = view[12:19]
print(chunk)
print('Size:          ', chunk.nbytes)
print('Data in view:   ', chunk.tobytes())
print('Underlying data:', chunk.obj)

>>>
<memory at 0x10951fb80>
Size:          7
Data in view:   b'haircut'
Underlying data: b'shave and a haircut, two bits'
```

By enabling *zero-copy* operations, `memoryview` can provide enormous speedups for code that needs to quickly process large amounts of memory, such as numerical C extensions like NumPy and I/O-bound programs like this one. Here, I replace the simple bytes slicing from above with `memoryview` slicing instead and repeat the same micro-benchmark:

[Click here to view code image](#)

```
video_view = memoryview(video_data)

def run_test():
    chunk = video_view[byte_offset:byte_offset + size]
    # Call socket.send(chunk), but ignoring for benchmark
    result = timeit.timeit(
        stmt='run_test()',
        globals=globals(),
        number=100) / 100

print(f'{result:0.9f} seconds')

>>>
0.000000250 seconds
```

The result is 250 nanoseconds. Now the theoretical maximum throughput of my server is 20 MB / 250 nanoseconds = 164 TB / second. For parallel clients, I can theoretically support up to 1 CPU-second / 250 nanoseconds = 4 million. That's more like it! This means that now my program is entirely

bound by the underlying performance of the socket connection to the client, not by CPU constraints.

Now, imagine that the data must flow in the other direction, where some clients are sending live video streams to the server in order to broadcast them to other users. In order to do this, I need to store the latest video data from the user in a cache that other clients can read from. Here's what the implementation of reading 1 MB of new data from the incoming client would look like:

[Click here to view code image](#)

```
socket = ...          # socket connection to the client
video_cache = ...      # Cache of incoming video stream
byte_offset = ...      # Incoming buffer position
size = 1024 * 1024    # Incoming chunk size

chunk = socket.recv(size)
video_view = memoryview(video_cache)
before = video_view[:byte_offset]
after = video_view[byte_offset + size:]
new_cache = b''.join([before, chunk, after])
```

The `socket.recv` method returns a bytes instance. I can splice the new data with the existing cache at the current `byte_offset` by using simple slicing operations and the `bytes.join` method. To understand the performance of this, I can run another micro-benchmark. I'm using a dummy socket, so the performance test is only for the memory operations, not the I/O interaction:

[Click here to view code image](#)

```
def run_test():
    chunk = socket.recv(size)
    before = video_view[:byte_offset]
    after = video_view[byte_offset + size:]
    new_cache = b''.join([before, chunk, after])
result = timeit.timeit(
    stmt='run_test()',
    globals=globals(),
    number=100) / 100

print(f'{result:0.9f} seconds')
```



```
>>>
0.033520550 seconds
```

It takes 33 milliseconds to receive 1 MB and update the video cache. This means my maximum receive throughput is 1 MB / 33 milliseconds = 31 MB / second, and I'm limited to 31 MB / 1 MB = 31 simultaneous clients streaming in video data this way. This doesn't scale.

A better way to write this code is to use Python's built-in bytearray type in conjunction with memoryview. One limitation with bytes instances is that they are read-only and don't allow for individual indexes to be updated:

[Click here to view code image](#)

```
my_bytes = b'hello'
my_bytes[0] = b'\x79'
```

```
>>>
Traceback ...
TypeError: 'bytes' object does not support item assignment
```

The bytearray type is like a mutable version of bytes that allows for arbitrary positions to be overwritten. bytearray uses integers for its values instead of bytes:

```
my_array = bytearray(b'hello')
my_array[0] = 0x79
print(my_array)
```

```
>>>
bytearray(b'yello')
```

A memoryview can also be used to wrap a bytearray. When you slice such a memoryview, the resulting object can be used to assign data to a particular portion of the underlying buffer. This eliminates the copying costs from above that were required to splice the bytes instances back together after data was received from the client:

[Click here to view code image](#)

```
my_array = bytearray(b'row, row, row your boat')
my_view = memoryview(my_array)
write_view = my_view[3:13]
write_view[:] = b'-10 bytes-'
```

```
print(my_array)
```

```
>>>
```

```
bytearray(b'row-10 bytes- your boat')
```

Many library methods in Python, such as `socket.recv_into` and `RawIOBase.readinto`, use the buffer protocol to receive or read data quickly. The benefit of these methods is that they avoid allocating memory and creating another copy of the data; what's received goes straight into an existing buffer. Here, I use `socket.recv_into` along with a `memoryview` slice to receive data into an underlying `bytearray` without the need for splicing:

[Click here to view code image](#)

```
video_array = bytearray(video_cache)
write_view = memoryview(video_array)
chunk = write_view[byte_offset:byte_offset + size]
socket.recv_into(chunk)
```

I can run another micro-benchmark to compare the performance of this approach to the earlier example that used `socket.recv`:

[Click here to view code image](#)

```
def run_test():
    chunk = write_view[byte_offset:byte_offset + size]
    socket.recv_into(chunk)

result = timeit.timeit(
    stmt='run_test()',
    globals=globals(),
    number=100) / 100

print(f'{result:0.9f} seconds')
```

```
>>>
```

```
0.000033925 seconds
```

It took 33 microseconds to receive a 1 MB video transmission. This means my server can support  $1 \text{ MB} / 33 \text{ microseconds} = 31 \text{ GB} / \text{second}$  of max throughput, and  $31 \text{ GB} / 1 \text{ MB} = 31,000$  parallel streaming clients. That's the type of scalability that I'm looking for!

## Things to Remember

- ✦ The `memoryview` built-in type provides a zero-copy interface for reading and writing slices of objects that support Python's highperformance buffer protocol.
- ✦ The `bytearray` built-in type provides a mutable bytes-like type that can be used for zero-copy data reads with functions like `socket.recv_from`.
- ✦ A `memoryview` can wrap a `bytearray`, allowing for received data to be spliced into an arbitrary buffer location without copying costs.

## 9. Testing and Debugging

Python doesn't have compile-time static type checking. There's nothing in the interpreter that will ensure that your program will work correctly when you run it. Python does support optional type annotations that can be used in static analysis to detect many kinds of bugs (see [Item 90: “Consider Static Analysis via typing to Obviate Bugs”](#) for details). However, it's still fundamentally a dynamic language, and anything is possible. With Python, you ultimately don't know if the functions your program calls will be defined at runtime, even when their existence is evident in the source code. This dynamic behavior is both a blessing and a curse.

The large numbers of Python programmers out there say it's worth going without compile-time static type checking because of the productivity gained from the resulting brevity and simplicity. But most people using Python have at least one horror story about a program encountering a boneheaded error at runtime. One of the worst examples I've heard of involved a `SyntaxError` being raised in production as a side effect of a dynamic import (see [Item 88: “Know How to Break Circular Dependencies”](#)), resulting in a crashed server process. The programmer I know who was hit by this surprising occurrence has since ruled out using Python ever again.

But I have to wonder, why wasn't the code more well tested before the program was deployed to production? Compile-time static type safety isn't everything. You should always test your code, regardless of what language it's written in. However, I'll admit that in Python it may be more important to write tests to verify correctness than in other languages. Luckily, the same dynamic features that create risks also make it extremely easy to write tests for your code and to debug malfunctioning programs. You can use Python's dynamic nature and easily overridable behaviors to implement tests and ensure that your programs work as expected.

You should think of tests as an insurance policy on your code. Good tests give you confidence that your code is correct. If you refactor or expand your code, tests that verify behavior—*not* implementation—make it easy to

identify what's changed. It sounds counterintuitive, but having good tests actually makes it easier to modify Python code, not harder.

## Item 75: Use `repr` Strings for Debugging Output

When debugging a Python program, the `print` function and format strings (see [Item 4: “Prefer Interpolated F-Strings Over C-style Format Strings and `str.format`”](#)), or output via the `logging` built-in module, will get you surprisingly far. Python internals are often easy to access via plain attributes (see [Item 42: “Prefer Public Attributes Over Private Ones”](#)). All you need to do is call `print` to see how the state of your program changes while it runs and understand where it goes wrong.

The `print` function outputs a human-readable string version of whatever you supply it. For example, printing a basic string prints the contents of the string without the surrounding quote characters:

```
print('foo bar')
```

```
>>>
foo bar
```

This is equivalent to all of these alternatives:

- Calling the `str` function before passing the value to `print`
- Using the `'%s'` format string with the `%` operator
- Default formatting of the value with an f-string
- Calling the `format` built-in function
- Explicitly calling the `__format__` special method
- Explicitly calling the `__str__` special method

Here, I verify this behavior:

```
my_value = 'foo bar'
print(str(my_value))
print('%s' % my_value)
print(f'{my_value}')
print(format(my_value))
```

```
print(my_value.__format__('s'))
print(my_value.__str__())
```

```
>>>
foo bar
foo bar
foo bar
foo bar
foo bar
foo bar
```

The problem is that the human-readable string for a value doesn't make it clear what the actual type and its specific composition are. For example, notice how in the default output of `print`, you can't distinguish between the types of the number 5 and the string '5':

[Click here to view code image](#)

```
print(5)
print('5')

int_value = 5
str_value = '5'
print(f'{int_value} == {str_value} ?')

>>>
5
5
5 == 5 ?
```

If you're debugging a program with `print`, these type differences matter. What you almost always want while debugging is to see the `repr` version of an object. The `repr` built-in function returns the *printable representation* of an object, which should be its most clearly understandable string representation. For most built-in types, the string returned by `repr` is a valid Python expression:

```
a = '\x07'
print(repr(a))

>>>
'\x07'
```

Passing the value from `repr` to the `eval` built-in function should result in the same Python object that you started with (and, of course, in practice you should only use `eval` with extreme caution):

```
b = eval(repr(a))
assert a == b
```

When you're debugging with `print`, you should call `repr` on a value before printing to ensure that any difference in types is clear:

```
print(repr(5))
print(repr('5'))
```

```
>>>
5
'5'
```

This is equivalent to using the `'%r'` format string with the `%` operator or an f-string with the `!r` type conversion:

[Click here to view code image](#)

```
print('%r' % 5)
print('%r' % '5')

int_value = 5
str_value = '5'
print(f'{int_value!r} != {str_value!r}')
```

```
>>>
5
'5'
5 != '5'
```

For instances of Python classes, the default human-readable string value is the same as the `repr` value. This means that passing an instance to `print` will do the right thing, and you don't need to explicitly call `repr` on it. Unfortunately, the default implementation of `repr` for object subclasses isn't especially helpful. For example, here I define a simple class and then print one of its instances:

[Click here to view code image](#)

```

class OpaqueClass:
    def __init__(self, x, y):
        self.x = x
        self.y = y

obj = OpaqueClass(1, 'foo')
print(obj)

>>>
<__main__.OpaqueClass object at 0x10963d6d0>

```

This output can't be passed to the `eval` function, and it says nothing about the instance fields of the object.

There are two solutions to this problem. If you have control of the class, you can define your own `__repr__` special method that returns a string containing the Python expression that re-creates the object. Here, I define that function for the class above:

[Click here to view code image](#)

```

class BetterClass:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __repr__(self):
        return f'BetterClass({self.x!r}, {self.y!r})'

```

Now the repr value is much more useful:

```

obj = BetterClass(2, 'bar')
print(obj)

>>>
BetterClass(2, 'bar')

```

When you don't have control over the class definition, you can reach into the object's instance dictionary, which is stored in the `__dict__` attribute. Here, I print out the contents of an `OpaqueClass` instance:

```

obj = OpaqueClass(4, 'baz')
print(obj.__dict__)

>>>
{'x': 4, 'y': 'baz'}

```



## Things to Remember

- ✦ Calling `print` on built-in Python types produces the humanreadable string version of a value, which hides type information.
- ✦ Calling `repr` on built-in Python types produces the printable string version of a value. These `repr` strings can often be passed to the `eval` built-in function to get back the original value.
- ✦ `%s` in format strings produces human-readable strings like `str.%r` produces printable strings like `repr`. F-strings produce humanreadable strings for replacement text expressions unless you specify the `!r` suffix.
- ✦ You can define the `__repr__` special method on a class to customize the printable representation of instances and provide more detailed debugging information.

## Item 76: Verify Related Behaviors in `TestCase` Subclasses

The canonical way to write tests in Python is to use the `unittest` built-in module. For example, say I have the following utility function defined in `utils.py` that I would like to verify works correctly across a variety of inputs:

[Click here to view code image](#)

```
# utils.py
def to_str(data):
    if isinstance(data, str):
        return data
    elif isinstance(data, bytes):
        return data.decode('utf-8')
    else:
        raise TypeError('Must supply str or bytes, '
                        'found: %r' % data)
```

To define tests, I create a second file named `test_utils.py` or `utils_test.py`—the naming scheme you prefer is a style choice—that contains tests for each behavior that I expect:

[Click here to view code image](#)

```
# utils_test.py
from unittest import TestCase, main
from utils import to_str

class UtilsTestCase(TestCase):
    def test_to_str_bytes(self):
        self.assertEqual('hello', to_str(b'hello'))

    def test_to_str_str(self):
        self.assertEqual('hello', to_str('hello'))

    def test_failing(self):
        self.assertEqual('incorrect', to_str('hello'))

if __name__ == '__main__':
    main()
```

Then, I run the test file using the Python command line. In this case, two of the test methods pass and one fails, with a helpful error message about what went wrong:

[Click here to view code image](#)

```
$ python3 utils_test.py
F..
=====
FAIL: test_failing (__main__.UtilsTestCase)
-----
Traceback (most recent call last):
  File "utils_test.py", line 15, in test_failing
    self.assertEqual('incorrect', to_str('hello'))
AssertionError: 'incorrect' != 'hello'
- incorrect
+ hello

-----
Ran 3 tests in 0.002s

FAILED (failures=1)
```

Tests are organized into `TestCase` subclasses. Each test case is a method beginning with the word `test`. If a test method runs without raising any kind of `Exception` (including `AssertionError` from `assert` statements), the

test is considered to have passed successfully. If one test fails, the `TestCase` subclass continues running the other test methods so you can get a full picture of how all your tests are doing instead of stopping at the first sign of trouble.

If you want to iterate quickly to fix or improve a specific test, you can run only that test method by specifying its path within the test module on the command line:

[Click here to view code image](#)

```
$ python3 utils_test.py UtilsTestCase.test_to_str_bytes
.  
-----  
Ran 1 test in 0.000s  
  
OK
```

You can also invoke the debugger from directly within test methods at specific breakpoints in order to dig more deeply into the cause of failures (see [Item 80: “Consider Interactive Debugging with `pdb`”](#) for how to do that).

The `TestCase` class provides helper methods for making assertions in your tests, such as `assertEqual` for verifying equality, `assertTrue` for verifying Boolean expressions, and many more (see `help(TestCase)` for the full list). These are better than the built-in `assert` statement because they print out all of the inputs and outputs to help you understand the exact reason the test is failing. For example, here I have the same test case written with and without using a helper assertion method:

[Click here to view code image](#)

```
# assert_test.py  
from unittest import TestCase, main  
from utils import to_str  
  
class AssertTestCase(TestCase):  
    def test_assert_helper(self):  
        expected = 12  
        found = 2 * 5  
        self.assertEqual(expected, found)
```

```

def test_assert_statement(self):
    expected = 12
    found = 2 * 5
    assert expected == found

if __name__ == '__main__':
    main()

```

Which of these failure messages seems more helpful to you?

[Click here to view code image](#)

```

$ python3 assert_test.py
FF
=====
FAIL: test_assert_helper (__main__.AssertTestCase)
-----
Traceback (most recent call last):
  File "assert_test.py", line 16, in test_assert_helper
    self.assertEqual(expected, found)
AssertionError: 12 != 10

=====
FAIL: test_assert_statement (__main__.AssertTestCase)
-----
Traceback (most recent call last):
  File "assert_test.py", line 11, in test_assert_statement
    assert expected == found
AssertionError

-----
Ran 2 tests in 0.001s

FAILED (failures=2)

```

There’s also an `assertRaises` helper method for verifying exceptions that can be used as a context manager in `with` statements (see [Item 66: “Consider `contextlib` and `with` Statements for Reusable `try/finally` Behavior](#)” for how that works). This appears similar to a `try/except` statement and makes it abundantly clear where the exception is expected to be raised:

[Click here to view code image](#)

```

# utils_error_test.py
from unittest import TestCase, main

```

```

from utils import to_str

class UtilsErrorTestCase(TestCase):

    def test_to_str_bad(self):
        with self.assertRaises(TypeError):
            to_str(object())

    def test_to_str_bad_encoding(self):
        with self.assertRaises(UnicodeDecodeError):
            to_str(b'\xfa\xfa')

if __name__ == '__main__':
    main()

```

You can define your own helper methods with complex logic in `TestCase` subclasses to make your tests more readable. Just ensure that your method names don't begin with the word `test`, or they'll be run as if they're test cases. In addition to calling `TestCase` assertion methods, these custom test helpers often use the `fail` method to clarify which assumption or invariant wasn't met. For example, here I define a custom test helper method for verifying the behavior of a generator:

[Click here to view code image](#)

```

# helper_test.py
from unittest import TestCase, main

def sum_squares(values):
    cumulative = 0
    for value in values:
        cumulative += value ** 2
    yield cumulative

class HelperTestCase(TestCase):
    def verify_complex_case(self, values, expected):
        expect_it = iter(expected)
        found_it = iter(sum_squares(values))
        test_it = zip(expect_it, found_it)

        for i, (expect, found) in enumerate(test_it):
            self.assertEqual(
                expect,
                found,
                f'Index {i} is wrong')

```

```

    # Verify both generators are exhausted
    try:
        next(expect_it)
    except StopIteration:
        pass
    else:
        self.fail('Expected longer than found')

    try:
        next(found_it)
    except StopIteration:
        pass
    else:
        self.fail('Found longer than expected')
def test_wrong_lengths(self):
    values = [1.1, 2.2, 3.3]
    expected = [
        1.1**2,
    ]
    self.verify_complex_case(values, expected)

def test_wrong_results(self):
    values = [1.1, 2.2, 3.3]
    expected = [
        1.1**2,
        1.1**2 + 2.2**2,
        1.1**2 + 2.2**2 + 3.3**2 + 4.4**2,
    ]
    self.verify_complex_case(values, expected)

if __name__ == '__main__':
    main()

```

The helper method makes the test cases short and readable, and the outputted error messages are easy to understand:

[Click here to view code image](#)

```

$ python3 helper_test.py
FF
=====
FAIL: test_wrong_lengths (__main__.HelperTestCase)
-----
Traceback (most recent call last):
  File "helper_test.py", line 43, in test_wrong_lengths
    self.verify_complex_case(values, expected)

```

```

    File "helper_test.py", line 34, in verify_complex_case
        self.fail('Found longer than expected')
AssertionError: Found longer than expected
=====
FAIL: test_wrong_results (__main__.HelperTestCase)
-----
Traceback (most recent call last):
  File "helper_test.py", line 52, in test_wrong_results
    self.verify_complex_case(values, expected)
  File "helper_test.py", line 24, in verify_complex_case
    f'Index {i} is wrong')
AssertionError: 36.3 != 16.939999999999998 : Index 2 is wrong

-----
Ran 2 tests in 0.002s

FAILED (failures=2)

```

I usually define one `TestCase` subclass for each set of related tests. Sometimes, I have one `TestCase` subclass for each function that has many edge cases. Other times, a `TestCase` subclass spans all functions in a single module. I often create one `TestCase` subclass for testing each basic class and all of its methods.

The `TestCase` class also provides a `subTest` helper method that enables you to avoid boilerplate by defining multiple tests within a single test method. This is especially helpful for writing data-driven tests, and it allows the test method to continue testing other cases even after one of them fails (similar to the behavior of `TestCase` with its contained test methods). To show this, here I define an example data-driven test:

[Click here to view code image](#)

```

# data_driven_test.py
from unittest import TestCase, main
from utils import to_str

class DataDrivenTestCase(TestCase):
    def test_good(self):
        good_cases = [
            (b'my bytes', 'my bytes'),
            ('no error', b'no error'), # This one will fail
            ('other str', 'other str'),
            ...

```

```

    ]
    for value, expected in good_cases:
        with self.subTest(value):
            self.assertEqual(expected, to_str(value))
    def test_bad(self):
        bad_cases = [
            (object(), TypeError),
            (b'\xfa\xfa', UnicodeDecodeError),
            ...
        ]
        for value, exception in bad_cases:
            with self.subTest(value):
                with self.assertRaises(exception):
                    to_str(value)

if __name__ == '__main__':
    main()

```

The 'no error' test case fails, printing a helpful error message, but all of the other cases are still tested and confirmed to pass:

[Click here to view code image](#)

```

$ python3 data_driven_test.py
.
=====
FAIL: test_good (__main__.DataDrivenTestCase) [no error]
-----
Traceback (most recent call last):
  File "testing/data_driven_test.py", line 18, in test_good
    self.assertEqual(expected, to_str(value))
AssertionError: b'no error' != 'no error'

-----
Ran 2 tests in 0.001s

FAILED (failures=1)

```

## Note

Depending on your project's complexity and testing requirements, the *pytest* (<https://pytest.org>) open source package and its large number of community plug-ins can be especially useful.

## Things to Remember



- ✦ You can create tests by subclassing the `TestCase` class from the `unittest` built-in module and defining one method per behavior you'd like to test. Test methods on `TestCase` classes must start with the word `test`.
- ✦ Use the various helper methods defined by the `TestCase` class, such as `assertEqual`, to confirm expected behaviors in your tests instead of using the built-in `assert` statement.
- ✦ Consider writing data-driven tests using the `subTest` helper method in order to reduce boilerplate.

## Item 77: Isolate Tests from Each Other with `setUp`, `tearDown`, `setUpModule`, and `tearDownModule`

`TestCase` classes (see [Item 76: “Verify Related Behaviors in `TestCase` Subclasses](#)”) often need to have the test environment set up before test methods can be run; this is sometimes called the *test harness*. To do this, you can override the `setUp` and `tearDown` methods of a `TestCase` subclass. These methods are called before and after each test method, respectively, so you can ensure that each test runs in isolation, which is an important best practice of proper testing.

For example, here I define a `TestCase` that creates a temporary directory before each test and deletes its contents after each test finishes:

[Click here to view code image](#)

```
# environment_test.py
from pathlib import Path
from tempfile import TemporaryDirectory
from unittest import TestCase, main

class EnvironmentTest(TestCase):
    def setUp(self):
        self.test_dir = TemporaryDirectory()
        self.test_path = Path(self.test_dir.name)

    def tearDown(self):
        self.test_dir.cleanup()
```

```

def test_modify_file(self):
    with open(self.test_path / 'data.bin', 'w') as f:
        ...

if __name__ == '__main__':
    main()

```

When programs get complicated, you'll want additional tests to verify the end-to-end interactions between your modules instead of only testing code in isolation (using tools like mocks; see [Item 78: “Use Mocks to Test Code with Complex Dependencies”](#)). This is the difference between *unit tests* and *integration tests*. In Python, it's important to write both types of tests for exactly the same reason: You have no guarantee that your modules will actually work together unless you prove it.

One common problem is that setting up your test environment for integration tests can be computationally expensive and may require a lot of wall-clock time. For example, you might need to start a database process and wait for it to finish loading indexes before you can run your integration tests. This type of latency makes it impractical to do test preparation and cleanup for every test in the `TestCase` class's `setUp` and `tearDown` methods.

To handle this situation, the `unittest` module also supports module-level test harness initialization. You can configure expensive resources a single time, and then have all `TestCase` classes and their test methods run without repeating that initialization. Later, when all tests in the module are finished, the test harness can be torn down a single time. Here, I take advantage of this behavior by defining `setUpModule` and `tearDownModule` functions within the module containing the `TestCase` classes:

[Click here to view code image](#)

```

# integration_test.py
from unittest import TestCase, main

def setUpModule():
    print('* Module setup')

def tearDownModule():
    print('* Module clean-up')

class IntegrationTest(TestCase):

```

```

def setUp(self):
    print('* Test setup')

def tearDown(self):
    print('* Test clean-up')

def test_end_to_end1(self):
    print('* Test 1')

def test_end_to_end2(self):
    print('* Test 2')

if __name__ == '__main__':
    main()

```

```

$ python3 integration_test.py
* Module setup
* Test setup
* Test 1
* Test clean-up
.* Test setup
* Test 2
* Test clean-up
.* Module clean-up

```

```

-----
Ran 2 tests in 0.000s

```

OK

I can clearly see that `setUpModule` is run by `unittest` only once, and it happens before any `setUp` methods are called. Similarly, `tearDownModule` happens after the `tearDown` method is called.

## Things to Remember

- ◆ It's important to write both unit tests (for isolated functionality) and integration tests (for modules that interact with each other).
- ◆ Use the `setUp` and `tearDown` methods to make sure your tests are isolated from each other and have a clean test environment.
- ◆ For integration tests, use the `setUpModule` and `tearDownModule` module-level functions to manage any test harnesses you need for the entire

lifetime of a test module and all of the `TestCase` classes that it contains.

## Item 78: Use Mocks to Test Code with Complex Dependencies

Another common need when writing tests (see [Item 76: “Verify Related Behaviors in `TestCase` Subclasses”](#)) is to use mocked functions and classes to simulate behaviors when it’s too difficult or slow to use the real thing. For example, say that I need a program to maintain the feeding schedule for animals at the zoo. Here, I define a function to query a database for all of the animals of a certain species and return when they most recently ate:

[Click here to view code image](#)

```
class DatabaseConnection:
    ...

def get_animals(database, species):
    # Query the database
    ...
    # Return a list of (name, last_mealtime) tuples
```

How do I get a `DatabaseConnection` instance to use for testing this function? Here, I try to create one and pass it into the function being tested:

[Click here to view code image](#)

```
database = DatabaseConnection('localhost', '4444')

get_animals(database, 'Meerkat')

>>>
Traceback ...
DatabaseConnectionError: Not connected
```

There’s no database running, so of course this fails. One solution is to actually stand up a database server and connect to it in the test. However, it’s a lot of work to fully automate starting up a database, configuring its schema, populating it with data, and so on in order to just run a simple unit test. Further, it will probably take a lot of wallclock time to set up a

database server, which would slow down these unit tests and make them harder to maintain.

A better approach is to mock out the database. A *mock* lets you provide expected responses for dependent functions, given a set of expected calls. It's important not to confuse mocks with fakes. A *fake* would provide most of the behavior of the `DatabaseConnection` class but with a simpler implementation, such as a basic in-memory, single-threaded database with no persistence.

Python has the `unittest.mock` built-in module for creating mocks and using them in tests. Here, I define a `Mock` instance that simulates the `get_animals` function without actually connecting to the database:

[Click here to view code image](#)

```
from datetime import datetime
from unittest.mock import Mock

mock = Mock(spec=get_animals)
expected = [
    ('Spot', datetime(2019, 6, 5, 11, 15)),
    ('Fluffy', datetime(2019, 6, 5, 12, 30)),
    ('Jojo', datetime(2019, 6, 5, 12, 45)),
]
mock.return_value = expected
```

The `Mock` class creates a mock function. The `return_value` attribute of the mock is the value to return when it is called. The `spec` argument indicates that the mock should act like the given object, which is a function in this case, and error if it's used in the wrong way.

For example, here I try to treat the mock function as if it were a mock object with attributes:

[Click here to view code image](#)

```
mock.does_not_exist

>>>
Traceback ...
AttributeError: Mock object has no attribute 'does_not_exist'
```

Once it's created, I can call the mock, get its return value, and verify that what it returns matches expectations. I use a unique object value as the database argument because it won't actually be used by the mock to do anything; all I care about is that the database parameter was correctly plumbed through to any dependent functions that needed a `DatabaseConnection` instance in order to work (see [Item 55: “Use queue to Coordinate Work Between Threads”](#) for another example of using sentinel object instances):

```
database = object()
result = mock(database, 'Meerkat')
assert result == expected
```

This verifies that the mock responded correctly, but how do I know if the code that called the mock provided the correct arguments? For this, the `Mock` class provides the `assert_called_once_with` method, which verifies that a single call with exactly the given parameters was made:

[Click here to view code image](#)

```
mock.assert_called_once_with(database, 'Meerkat')
```

If I supply the wrong parameters, an exception is raised, and any `TestCase` that the assertion is used in fails:

[Click here to view code image](#)

```
mock.assert_called_once_with(database, 'Giraffe')
```

```
>>>
Traceback ...
AssertionError: expected call not found.
Expected: mock(<object object at 0x109038790>, 'Giraffe')
Actual: mock(<object object at 0x109038790>, 'Meerkat')
```

If I actually don't care about some of the individual parameters, such as exactly which database object was used, then I can indicate that any value is okay for an argument by using the `unittest.mock.ANY` constant. I can also use the `assert_called_with` method of `Mock` to verify that the most recent call to the mock—and there may have been multiple calls in this case—matches my expectations:

[Click here to view code image](#)

```
from unittest.mock import ANY
mock = Mock(spec=get_animals)
mock('database 1', 'Rabbit')
mock('database 2', 'Bison')
mock('database 3', 'Meerkat')

mock.assert_called_with(ANY, 'Meerkat')
```

ANY is useful in tests when a parameter is not core to the behavior that's being tested. It's often worth erring on the side of under-specifying tests by using ANY more liberally instead of over-specifying tests and having to plumb through various test parameter expectations.

The Mock class also makes it easy to mock exceptions being raised:

[Click here to view code image](#)

```
class MyError(Exception):
    pass

mock = Mock(spec=get_animals)
mock.side_effect = MyError('Whoops! Big problem')
result = mock(database, 'Meerkat')

>>>
Traceback ...
MyError: Whoops! Big problem
```

There are many more features available, so be sure to see `help(unittest.mock.Mock)` for the full range of options.

Now that I've shown the mechanics of how a Mock works, I can apply it to an actual testing situation to show how to use it effectively in writing unit tests. Here, I define a function to do the rounds of feeding animals at the zoo, given a set of database-interacting functions:

[Click here to view code image](#)

```
def get_food_period(database, species):
    # Query the database
    ...
    # Return a time delta
```

```

def feed_animal(database, name, when):
    # Write to the database
    ...

def do_rounds(database, species):
    now = datetime.datetime.utcnow()
    feeding_timedelta = get_food_period(database, species)
    animals = get_animals(database, species)
    fed = 0
    for name, last_mealtime in animals:
        if (now - last_mealtime) > feeding_timedelta:
            feed_animal(database, name, now)
            fed += 1

    return fed

```

The goal of my test is to verify that when `do_rounds` is run, the right animals got fed, the latest feeding time was recorded to the database, and the total number of animals fed returned by the function matches the correct total. In order to do all this, I need to mock out `datetime.utcnow` so my tests have a stable time that isn't affected by daylight saving time and other ephemeral changes. I need to mock out `get_food_period` and `get_animals` to return values that would have come from the database. And I need to mock out `feed_animal` to accept data that would have been written back to the database.

The question is: Even if I know how to create these mock functions and set expectations, how do I get the `do_rounds` function that's being tested to use the mock dependent functions instead of the real versions? One approach is to inject everything as keyword-only arguments (see [Item 25: “Enforce Clarity with Keyword-Only and Positional-Only Arguments”](#)):

[Click here to view code image](#)

```

def do_rounds(database, species, *,
              now_func=datetime.utcnow,
              food_func=get_food_period,
              animals_func=get_animals,
              feed_func=feed_animal):
    now = now_func()
    feeding_timedelta = food_func(database, species)
    animals = animals_func(database, species)
    fed = 0

```



```

for name, last_mealtime in animals:
    if (now - last_mealtime) > feeding_timedelta:
        feed_func(database, name, now)
        fed += 1

return fed

```

To test this function, I need to create all of the Mock instances upfront and set their expectations:

[Click here to view code image](#)

```

from datetime import timedelta

now_func = Mock(spec=datetime.utcnow)
now_func.return_value = datetime(2019, 6, 5, 15, 45)

food_func = Mock(spec=get_food_period)
food_func.return_value = timedelta(hours=3)

animals_func = Mock(spec=get_animals)
animals_func.return_value = [
    ('Spot', datetime(2019, 6, 5, 11, 15)),
    ('Fluffy', datetime(2019, 6, 5, 12, 30)),
    ('Jojo', datetime(2019, 6, 5, 12, 45)),
]

feed_func = Mock(spec=feed_animal)

```

Then, I can run the test by passing the mocks into the do\_rounds function to override the defaults:

```

result = do_rounds(
    database,
    'Meerkat',
    now_func=now_func,
    food_func=food_func,
    animals_func=animals_func,
    feed_func=feed_func)

assert result == 2

```

Finally, I can verify that all of the calls to dependent functions matched my expectations:

[Click here to view code image](#)

```
from unittest.mock import call

food_func.assert_called_once_with(database, 'Meerkat')

animals_func.assert_called_once_with(database, 'Meerkat')

feed_func.assert_has_calls([
    call(database, 'Spot', now_func.return_value),
    call(database, 'Fluffy', now_func.return_value),
],
    any_order=True)
```

I don't verify the parameters to the `datetime.utcnow` mock or how many times it was called because that's indirectly verified by the return value of the function. For `get_food_period` and `get_animals`, I verify a single call with the specified parameters by using `assert_called_once_with`. For the `feed_animal` function, I verify that two calls were made—and their order didn't matter—to write to the database using the `unittest.mock.call` helper and the `assert_has_calls` method.

This approach of using keyword-only arguments for injecting mocks works, but it's quite verbose and requires changing every function you want to test. The `unittest.mock.patch` family of functions makes injecting mocks easier. It temporarily reassigns an attribute of a module or class, such as the database-accessing functions that I defined above. For example, here I override `get_animals` to be a mock using `patch`:

[Click here to view code image](#)

```
from unittest.mock import patch

print('Outside patch:', get_animals)

with patch('__main__.get_animals'):
    print('Inside patch: ', get_animals)

print('Outside again:', get_animals)
>>>
Outside patch: <function get_animals at 0x109217040>
Inside patch: <MagicMock name='get_animals' id='4454622832'>
Outside again: <function get_animals at 0x109217040>
```

patch works for many modules, classes, and attributes. It can be used in with statements (see [Item 66: “Consider contextlib and with Statements for Reusable try/finally Behavior”](#)), as a function decorator (see [Item 26: “Define Function Decorators with functools.wraps”](#)), or in the setUp and tearDown methods of TestCase classes (see [Item 76: “Verify Related Behaviors in TestCase Subclasses”](#)). For the full range of options, see `help(unittest.mock.patch)`.

However, patch doesn’t work in all cases. For example, to test `do_rounds` I need to mock out the current time returned by the `datetime.utcnow` class method. Python won’t let me do that because the `datetime` class is defined in a C-extension module, which can’t be modified in this way:

[Click here to view code image](#)

```
fake_now = datetime(2019, 6, 5, 15, 45)

with patch('datetime.datetime.utcnow'):
    datetime.utcnow.return_value = fake_now

>>>
Traceback ...
TypeError: can't set attributes of built-in/extension type
➔ 'datetime.datetime'
```

To work around this, I can create another helper function to fetch time that can be patched:

[Click here to view code image](#)

```
def get_do_rounds_time():
    return datetime.datetime.utcnow()

def do_rounds(database, species):
    now = get_do_rounds_time()
    ...

with patch('__main__.get_do_rounds_time'):
    ...
```

Alternatively, I can use a keyword-only argument for the `datetime.utcnow` mock and use patch for all of the other mocks:

[Click here to view code image](#)

```
def do_rounds(database, species, *, utcnow=datetime.utcnow):
    now = utcnow()
    feeding_timedelta = get_food_period(database, species)
    animals = get_animals(database, species)
    fed = 0

    for name, last_mealtime in animals:
        if (now - last_mealtime) > feeding_timedelta:
            feed_func(database, name, now)
            fed += 1

    return fed
```

I'm going to go with the latter approach. Now, I can use the `patch.multiple` function to create many mocks and set their expectations:

[Click here to view code image](#)

```
from unittest.mock import DEFAULT

with patch.multiple('__main__',
                    autospec=True,
                    get_food_period=DEFAULT,
                    get_animals=DEFAULT,
                    feed_animal=DEFAULT):
    now_func = Mock(spec=datetime.utcnow)
    now_func.return_value = datetime(2019, 6, 5, 15, 45)
    get_food_period.return_value = timedelta(hours=3)
    get_animals.return_value = [
        ('Spot', datetime(2019, 6, 5, 11, 15)),
        ('Fluffy', datetime(2019, 6, 5, 12, 30)),
        ('Jojo', datetime(2019, 6, 5, 12, 45))
    ]
```

With the setup ready, I can run the test and verify that the calls were correct inside the `with` statement that used `patch.multiple`:

[Click here to view code image](#)

```
result = do_rounds(database, 'Meerkat', utcnow=now_func)
assert result == 2

food_func.assert_called_once_with(database, 'Meerkat')
animals_func.assert_called_once_with(database, 'Meerkat')
feed_func.assert_has_calls([
```

```
[
    call(database, 'Spot', now_func.return_value),
    call(database, 'Fluffy', now_func.return_value),
],
any_order=True)
```

The keyword arguments to `patch.multiple` correspond to names in the `__main__` module that I want to override during the test. The `DEFAULT` value indicates that I want a standard `Mock` instance to be created for each name. All of the generated mocks will adhere to the specification of the objects they are meant to simulate, thanks to the `autospec=True` parameter.

These mocks work as expected, but it's important to realize that it's possible to further improve the readability of these tests and reduce boilerplate by refactoring your code to be more testable (see [Item 79: “Encapsulate Dependencies to Facilitate Mocking and Testing”](#)).

## Things to Remember

- ✦ The `unittest.mock` module provides a way to simulate the behavior of interfaces using the `Mock` class. Mocks are useful in tests when it's difficult to set up the dependencies that are required by the code that's being tested.
- ✦ When using mocks, it's important to verify both the behavior of the code being tested and how dependent functions were called by that code, using the `Mock.assert_called_once_with` family of methods.
- ✦ Keyword-only arguments and the `unittest.mock.patch` family of functions can be used to inject mocks into the code being tested.

## Item 79: Encapsulate Dependencies to Facilitate Mocking and Testing

In the previous item (see [Item 78: “Use Mocks to Test Code with Complex Dependencies”](#)), I showed how to use the facilities of the `unittest.mock` built-in module—including the `Mock` class and `patch` family of functions—to write tests that have complex dependencies, such as a database. However,

the resulting test code requires a lot of boilerplate, which could make it more difficult for new readers of the code to understand what the tests are trying to verify.

One way to improve these tests is to use a wrapper object to encapsulate the database's interface instead of passing a `DatabaseConnection` object to functions as an argument. It's often worth refactoring your code (see [Item 89: “Consider warnings to Refactor and Migrate Usage”](#) for one approach) to use better abstractions because it facilitates creating mocks and writing tests. Here, I redefine the various database helper functions from the previous item as methods on a class instead of as independent functions:

[Click here to view code image](#)

```
class ZooDatabase:
    ...

    def get_animals(self, species):
        ...

    def get_food_period(self, species):
        ...

    def feed_animal(self, name, when):
        ...
```

Now, I can redefine the `do_rounds` function to call methods on a `ZooDatabase` object:

[Click here to view code image](#)

```
from datetime import datetime

def do_rounds(database, species, *, utcnow=datetime.utcnow):

    now = utcnow()
    feeding_timedelta = database.get_food_period(species)
    animals = database.get_animals(species)
    fed = 0

    for name, last_mealtime in animals:
        if (now - last_mealtime) >= feeding_timedelta:
            database.feed_animal(name, now)
            fed += 1
```

```
return fed
```

Writing a test for `do_rounds` is now a lot easier because I no longer need to use `unittest.mock.patch` to inject the mock into the code being tested. Instead, I can create a `Mock` instance to represent a `ZooDatabase` and pass that in as the database parameter. The `Mock` class returns a mock object for any attribute name that is accessed. Those attributes can be called like methods, which I can then use to set expectations and verify calls. This makes it easy to mock out all of the methods of a class:

[Click here to view code image](#)

```
from unittest.mock import Mock

database = Mock(spec=ZooDatabase)
print(database.feed_animal)
database.feed_animal()
database.feed_animal.assert_any_call()

>>>
<Mock name='mock.feed_animal' id='4384773408'>
```

I can rewrite the `Mock` setup code by using the `ZooDatabase` encapsulation:

[Click here to view code image](#)

```
from datetime import timedelta
from unittest.mock import call

now_func = Mock(spec=datetime.utcnow)
now_func.return_value = datetime(2019, 6, 5, 15, 45)

database = Mock(spec=ZooDatabase)
database.get_food_period.return_value = timedelta(hours=3)
database.get_animals.return_value = [
    ('Spot', datetime(2019, 6, 5, 11, 15)),
    ('Fluffy', datetime(2019, 6, 5, 12, 30)),
    ('Jojo', datetime(2019, 6, 5, 12, 55))
]
```

Then I can run the function being tested and verify that all dependent methods were called as expected:

[Click here to view code image](#)

```

result = do_rounds(database, 'Meerkat', utcnow=now_func)
assert result == 2

database.get_food_period.assert_called_once_with('Meerkat')
database.get_animals.assert_called_once_with('Meerkat')
database.feed_animal.assert_has_calls(
    [
        call('Spot', now_func.return_value),
        call('Fluffy', now_func.return_value),
    ],
    any_order=True)

```

Using the `spec` parameter to `Mock` is especially useful when mocking classes because it ensures that the code under test doesn't call a misspelled method name by accident. This allows you to avoid a common pitfall where the same bug is present in both the code and the unit test, masking a real error that will later reveal itself in production:

[Click here to view code image](#)

```

database.bad_method_name()

>>>
Traceback ...
AttributeError: Mock object has no attribute 'bad_method_name'

```

If I want to test this program end-to-end with a mid-level integration test (see [Item 77: “Isolate Tests from Each Other with `setUp`, `tearDown`, `setUpModule`, and `tearDownModule`”](#)), I still need a way to inject a mock `ZooDatabase` into the program. I can do this by creating a helper function that acts as a seam for *dependency injection*. Here, I define such a helper function that caches a `ZooDatabase` in module scope (see [Item 86: “Consider Module-Scoped Code to Configure Deployment Environments”](#)) by using a `global` statement:

[Click here to view code image](#)

```

DATABASE = None

def get_database():
    global DATABASE
    if DATABASE is None:
        DATABASE = ZooDatabase()
    return DATABASE

```



```
def main(argv):
    database = get_database()
    species = argv[1]
    count = do_rounds(database, species)
    print(f'Fed {count} {species}(s)')
    return 0
```

Now, I can inject the mock ZooDatabase using patch, run the test, and verify the program's output. I'm not using a mock `datetime.utcnow` here; instead, I'm relying on the database records returned by the mock to be relative to the current time in order to produce similar behavior to the unit test. This approach is more flaky than mocking everything, but it also tests more surface area:

[Click here to view code image](#)

```
import contextlib
import io
from unittest.mock import patch
with patch('__main__.DATABASE', spec=ZooDatabase):
    now = datetime.utcnow()

    DATABASE.get_food_period.return_value = timedelta(hours=3)
    DATABASE.get_animals.return_value = [
        ('Spot', now - timedelta(minutes=4.5)),
        ('Fluffy', now - timedelta(hours=3.25)),
        ('Jojo', now - timedelta(hours=3)),
    ]

    fake_stdout = io.StringIO()
    with contextlib.redirect_stdout(fake_stdout):
        main(['program name', 'Meerkat'])

    found = fake_stdout.getvalue()
    expected = 'Fed 2 Meerkat(s)\n'

    assert found == expected
```

The results match my expectations. Creating this integration test was straightforward because I designed the implementation to make it easier to test.

## Things to Remember

- ◆ When unit tests require a lot of repeated boilerplate to set up mocks, one solution may be to encapsulate the functionality of dependencies into classes that are more easily mocked.
- ◆ The `Mock` class of the `unittest.mock` built-in module simulates classes by returning a new mock, which can act as a mock method, for each attribute that is accessed.
- ◆ For end-to-end tests, it's valuable to refactor your code to have more helper functions that can act as explicit seams to use for injecting mock dependencies in tests.

## Item 80: Consider Interactive Debugging with `pdb`

Everyone encounters bugs in code while developing programs. Using the `print` function can help you track down the sources of many issues (see [Item 75: “Use repr Strings for Debugging Output”](#)). Writing tests for specific cases that cause trouble is another great way to isolate problems (see [Item 76: “Verify Related Behaviors in TestCase Subclasses”](#)).

But these tools aren't enough to find every root cause. When you need something more powerful, it's time to try Python's built-in *interactive debugger*. The debugger lets you inspect program state, print local variables, and step through a Python program one statement at a time.

In most other programming languages, you use a debugger by specifying what line of a source file you'd like to stop on, and then execute the program. In contrast, with Python, the easiest way to use the debugger is by modifying your program to directly initiate the debugger just before you think you'll have an issue worth investigating. This means that there is no difference between starting a Python program in order to run the debugger and starting it normally.

To initiate the debugger, all you have to do is call the `breakpoint` built-in function. This is equivalent to importing the `pdb` built-in module and running its `set_trace` function:

[Click here to view code image](#)

```
# always_breakpoint.py
import math

def compute_rmse(observed, ideal):
    total_err_2 = 0
    count = 0
    for got, wanted in zip(observed, ideal):
        err_2 = (got - wanted) ** 2
        breakpoint() # Start the debugger here
        total_err_2 += err_2
        count += 1

    mean_err = total_err_2 / count
    rmse = math.sqrt(mean_err)
    return rmse

result = compute_rmse(
    [1.8, 1.7, 3.2, 6],
    [2, 1.5, 3, 5])
print(result)
```

As soon as the breakpoint function runs, the program pauses its execution before the line of code immediately following the breakpoint call. The terminal that started the program turns into an interactive Python shell:

[Click here to view code image](#)

```
$ python3 always_breakpoint.py
> always_breakpoint.py(12)compute_rmse()
-> total_err_2 += err_2
(Pdb)
```

At the (Pdb) prompt, you can type in the names of local variables to see their values printed out (or use `p <name>`). You can see a list of all local variables by calling the `locals` built-in function. You can import modules, inspect global state, construct new objects, run the `help` built-in function, and even modify parts of the running program—whatever you need to do to aid in your debugging.

In addition, the debugger has a variety of special commands to control and understand program execution; type `help` to see the full list.

Three very useful commands make inspecting the running program easier:

- `where`: Print the current execution call stack. This lets you figure out where you are in your program and how you arrived at the breakpoint trigger.
- `up`: Move your scope up the execution call stack to the caller of the current function. This allows you to inspect the local variables in higher levels of the program that led to the breakpoint.
- `down`: Move your scope back down the execution call stack one level.

When you're done inspecting the current state, you can use these five debugger commands to control the program's execution in different ways:

- `step`: Run the program until the next line of execution in the program, and then return control back to the debugger prompt. If the next line of execution includes calling a function, the debugger stops within the function that was called.
- `next`: Run the program until the next line of execution in the current function, and then return control back to the debugger prompt. If the next line of execution includes calling a function, the debugger will not stop until the called function has returned.
- `return`: Run the program until the current function returns, and then return control back to the debugger prompt.
- `continue`: Continue running the program until the next breakpoint is hit (either through the breakpoint call or one added by a debugger command).
- `quit`: Exit the debugger and end the program. Run this command if you've found the problem, gone too far, or need to make program modifications and try again.

The breakpoint function can be called anywhere in a program. If you know that the problem you're trying to debug happens only under special circumstances, then you can just write plain old Python code to call `breakpoint` after a specific condition is met. For example, here I start the debugger only if the squared error for a datapoint is more than 1:

[Click here to view code image](#)

```
# conditional_breakpoint.py
def compute_rmse(observed, ideal):
    ...

    for got, wanted in zip(observed, ideal):
        err_2 = (got - wanted) ** 2
        if err_2 >= 1: # Start the debugger if True
            breakpoint()
        total_err_2 += err_2
        count += 1

    ...

result = compute_rmse(
    [1.8, 1.7, 3.2, 7],
    [2, 1.5, 3, 5])
print(result)
```

When I run the program and it enters the debugger, I can confirm that the condition was true by inspecting local variables:

[Click here to view code image](#)

```
$ python3 conditional_breakpoint.py
> conditional_breakpoint.py(14)compute_rmse()
-> total_err_2 += err_2
(Pdb) wanted
5
(Pdb) got
7
(Pdb) err_2
4
```

Another useful way to reach the debugger prompt is by using *post-mortem debugging*. This enables you to debug a program *after* it's already raised an exception and crashed. This is especially helpful when you're not quite sure where to put the breakpoint function call.

Here, I have a script that will crash due to the 7j complex number being present in one of the function's arguments:

[Click here to view code image](#)

```
# postmortem_breakpoint.py
import math
```

```
def compute_rmse(observed, ideal):
    ...

result = compute_rmse(
    [1.8, 1.7, 3.2, 7j], # Bad input
    [2, 1.5, 3, 5])
print(result)
```

I use the command line `python3 -m pdb -c continue <program path>` to run the program under control of the `pdb` module. The `continue` command tells `pdb` to get the program started immediately. Once it's running, the program hits a problem and automatically enters the interactive debugger, at which point I can inspect the program state:

[Click here to view code image](#)

```
$ python3 -m pdb -c continue postmortem_breakpoint.py
Traceback (most recent call last):
  File ".../pdb.py", line 1697, in main
    pdb._runscript(mainpyfile)
  File ".../pdb.py", line 1566, in _runscript
    self.run(statement)
  File ".../bdb.py", line 585, in run
    exec(cmd, globals, locals)
  File "<string>", line 1, in <module>
  File "postmortem_breakpoint.py", line 4, in <module>
    import math
  File "postmortem_breakpoint.py", line 16, in compute_rmse
    rmse = math.sqrt(mean_err)
TypeError: can't convert complex to float
Uncaught exception. Entering post mortem debugging
Running 'cont' or 'step' will restart the program
> postmortem_breakpoint.py(16)compute_rmse()
-> rmse = math.sqrt(mean_err)
(Pdb) mean_err
(-5.97-17.5j)
```

You can also use post-mortem debugging after hitting an uncaught exception in the interactive Python interpreter by calling the `pm` function of the `pdb` module (which is often done in a single line as `import pdb; pdb.pm()`):

[Click here to view code image](#)

```

$ python3
>>> import my_module
>>> my_module.compute_stddev([5])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "my_module.py", line 17, in compute_stddev
    variance = compute_variance(data)
  File "my_module.py", line 13, in compute_variance
    variance = err_2_sum / (len(data) - 1)
ZeroDivisionError: float division by zero
>>> import pdb; pdb.pm()
> my_module.py(13)compute_variance()
-> variance = err_2_sum / (len(data) - 1)
(Pdb) err_2_sum
0.0
(Pdb) len(data)
1

```

## Things to Remember

- ✦ You can initiate the Python interactive debugger at a point of interest directly in your program by calling the breakpoint built-in function.
- ✦ The Python debugger prompt is a full Python shell that lets you inspect and modify the state of a running program.
- ✦ pdb shell commands let you precisely control program execution and allow you to alternate between inspecting program state and progressing program execution.
- ✦ The pdb module can be used for debug exceptions after they happen in independent Python programs (using `python -m pdb -c continue <program path>`) or the interactive Python interpreter (using `import pdb; pdb.pm()`).

## Item 81: Use `tracemalloc` to Understand Memory Usage and Leaks

Memory management in the default implementation of Python, CPython, uses reference counting. This ensures that as soon as all references to an object have expired, the referenced object is also cleared from memory,

freeing up that space for other data. CPython also has a built-in cycle detector to ensure that self-referencing objects are eventually garbage collected.

In theory, this means that most Python programmers don't have to worry about allocating or deallocating memory in their programs. It's taken care of automatically by the language and the CPython runtime. However, in practice, programs eventually do run out of memory due to no longer useful references still being held. Figuring out where a Python program is using or leaking memory proves to be a challenge.

The first way to debug memory usage is to ask the gc built-in module to list every object currently known by the garbage collector. Although it's quite a blunt tool, this approach lets you quickly get a sense of where your program's memory is being used.

Here, I define a module that fills up memory by keeping references:

[Click here to view code image](#)

```
# waste_memory.py
import os

class MyObject:
    def __init__(self):
        self.data = os.urandom(100)

def get_data():
    values = []
    for _ in range(100):
        obj = MyObject()
        values.append(obj)
    return values

def run():
    deep_values = []
    for _ in range(100):
        deep_values.append(get_data())
    return deep_values
```

Then, I run a program that uses the gc built-in module to print out how many objects were created during execution, along with a small sample of allocated objects:



[Click here to view code image](#)

```
# using_gc.py
import gc

found_objects = gc.get_objects()
print('Before:', len(found_objects))

import waste_memory

hold_reference = waste_memory.run()

found_objects = gc.get_objects()
print('After: ', len(found_objects))
for obj in found_objects[:3]:
    print(repr(obj)[:100])

>>>
Before: 6207
After: 16801

<waste_memory.MyObject object at 0x10390aeb8>
<waste_memory.MyObject object at 0x10390aef0>
<waste_memory.MyObject object at 0x10390af28>
...
```

The problem with `gc.get_objects` is that it doesn't tell you anything about *how* the objects were allocated. In complicated programs, objects of a specific class could be allocated many different ways. Knowing the overall number of objects isn't nearly as important as identifying the code responsible for allocating the objects that are leaking memory.

Python 3.4 introduced a new `tracemalloc` built-in module for solving this problem. `tracemalloc` makes it possible to connect an object back to where it was allocated. You use it by taking before and after snapshots of memory usage and comparing them to see what's changed. Here, I use this approach to print out the top three memory usage offenders in a program:

[Click here to view code image](#)

```
# top_n.py
import tracemalloc

tracemalloc.start(10)                                     # Set stack depth
time1 = tracemalloc.take_snapshot()                       # Before snapshot
```

```
import waste_memory

x = waste_memory.run()
time2 = tracemalloc.take_snapshot()

stats = time2.compare_to(time1, 'lineno')
for stat in stats[:3]:
    print(stat)
```

```
>>>
waste_memory.py:5: size=2392 KiB (+2392 KiB), count=29994
➔(+29994), average=82 B
waste_memory.py:10: size=547 KiB (+547 KiB), count=10001
➔(+10001), average=56 B
waste_memory.py:11: size=82.8 KiB (+82.8 KiB), count=100
➔(+100), average=848 B
```

The size and count labels in the output make it immediately clear which objects are dominating my program's memory usage and where in the source code they were allocated.

The `tracemalloc` module can also print out the full stack trace of each allocation (up to the number of frames passed to the `tracemalloc.start` function). Here, I print out the stack trace of the biggest source of memory usage in the program:

[Click here to view code image](#)

```
# with_trace.py
import tracemalloc

tracemalloc.start(10)
time1 = tracemalloc.take_snapshot()

import waste_memory

x = waste_memory.run()
time2 = tracemalloc.take_snapshot()

stats = time2.compare_to(time1, 'traceback')
top = stats[0]
print('Biggest offender is:')
print('\n'.join(top.traceback.format()))

>>>
```

Biggest offender is:

```
File "with_trace.py", line 9
    x = waste_memory.run()
File "waste_memory.py", line 17
    deep_values.append(get_data())
File "waste_memory.py", line 10
    obj = MyObject()
File "waste_memory.py", line 5
    self.data = os.urandom(100)
```

A stack trace like this is most valuable for figuring out which particular usage of a common function or class is responsible for memory consumption in a program.

## Things to Remember

- ✦ It can be difficult to understand how Python programs use and leak memory.
- ✦ The `gc` module can help you understand which objects exist, but it has no information about how they were allocated.
- ✦ The `tracemalloc` built-in module provides powerful tools for understanding the sources of memory usage.

## 10. Collaboration

Python has language features that help you construct well-defined APIs with clear interface boundaries. The Python community has established best practices to maximize the maintainability of code over time. In addition, some standard tools that ship with Python enable large teams to work together across disparate environments.

Collaborating with others on Python programs requires being deliberate in how you write your code. Even if you're working on your own, chances are you'll be using code written by someone else via the standard library or open source packages. It's important to understand the mechanisms that make it easy to collaborate with other Python programmers.

### Item 82: Know Where to Find Community-Built Modules

Python has a central repository of modules (<https://pypi.org>) that you can install and use in your programs. These modules are built and maintained by people like you: the Python community. When you find yourself facing an unfamiliar challenge, the Python Package Index (PyPI) is a great place to look for code that will get you closer to your goal.

To use the Package Index, you need to use the command-line tool `pip` (a recursive acronym for “pip installs packages”). `pip` can be run with `python3 -m pip` to ensure that packages are installed for the correct version of Python on your system (see [Item 1: “Know Which Version of Python You’re Using”](#)). Using `pip` to install a new module is simple. For example, here I install the `pytz` module that I use elsewhere in this book (see [Item 67: “Use `datetime` Instead of `time` for Local Clocks”](#)):

```
$ python3 -m pip install pytz
Collecting pytz
  Downloading ...
Installing collected packages: pytz
Successfully installed pytz-2018.9
```

pip is best used together with the built-in module `venv` to consistently track sets of packages to install for your projects (see [Item 83: “Use Virtual Environments for Isolated and Reproducible Dependencies”](#)). You can also create your own PyPI packages to share with the Python community or host your own private package repositories for use with pip.

Each module in the PyPI has its own software license. Most of the packages, especially the popular ones, have free or open source licenses (see <https://opensource.org> for details). In most cases, these licenses allow you to include a copy of the module with your program; when in doubt, talk to a lawyer.

## Things to Remember

- ◆ The Python Package Index (PyPI) contains a wealth of common packages that are built and maintained by the Python community.
- ◆ pip is the command-line tool you can use to install packages from PyPI.
- ◆ The majority of PyPI modules are free and open source software.

## Item 83: Use Virtual Environments for Isolated and Reproducible Dependencies

Building larger and more complex programs often leads you to rely on various packages from the Python community (see [Item 82: “Know Where to Find Community-Built Modules”](#)). You’ll find yourself running the `python3 -m pip` command-line tool to install packages like `pytz`, `numpy`, and many others.

The problem is that, by default, pip installs new packages in a global location. That causes all Python programs on your system to be affected by these installed modules. In theory, this shouldn’t be an issue. If you install a package and never import it, how could it affect your programs?

The trouble comes from transitive dependencies: the packages that the packages you install depend on. For example, you can see what the `sphinx`

package depends on after installing it by asking pip:

[Click here to view code image](#)

```
$ python3 -m pip show Sphinx
Name: Sphinx
Version: 2.1.2
Summary: Python documentation generator
Location: /usr/local/lib/python3.8/site-packages
Requires: alabaster, imagesize, requests,
➔ sphinxcontrib-applehelp, sphinxcontrib-qthelp,
➔ Jinja2, setuptools, sphinxcontrib-jsmath,
➔ sphinxcontrib-serializinghtml, Pygments, snowballstemmer,
➔ packaging, sphinxcontrib-devhelp, sphinxcontrib-htmlhelp,
➔ babel, docutils
Required-by:
```

If you install another package like `flask`, you can see that it, too, depends on the `Jinja2` package:

[Click here to view code image](#)

```
$ python3 -m pip show flask
Name: Flask
Version: 1.0.3
Summary: A simple framework for building complex web applications.
Location: /usr/local/lib/python3.8/site-packages
Requires: itsdangerous, click, Jinja2, Werkzeug
Required-by:
```

A dependency conflict can arise as `Sphinx` and `flask` diverge over time. Perhaps right now they both require the same version of `Jinja2`, and everything is fine. But six months or a year from now, `Jinja2` may release a new version that makes breaking changes to users of the library. If you update your global version of `Jinja2` with `python3 -m pip install --upgrade Jinja2`, you may find that `Sphinx` breaks, while `flask` keeps working.

The cause of such breakage is that Python can have only a single global version of a module installed at a time. If one of your installed packages must use the new version and another package must use the old version, your system isn't going to work properly; this situation is often called *dependency hell*.

Such breakage can even happen when package maintainers try their best to preserve API compatibility between releases (see [Item 85: “Use Packages to Organize Modules and Provide Stable APIs”](#)). New versions of a library can subtly change behaviors that API-consuming code relies on. Users on a system may upgrade one package to a new version but not others, which could break dependencies. If you’re not careful there’s a constant risk of the ground moving beneath your feet.

These difficulties are magnified when you collaborate with other developers who do their work on separate computers. It’s best to assume the worst: that the versions of Python and global packages that they have installed on their machines will be slightly different from yours. This can cause frustrating situations such as a codebase working perfectly on one programmer’s machine and being completely broken on another’s.

The solution to all of these problems is using a tool called `venv`, which provides *virtual environments*. Since Python 3.4, `pip` and the `venv` module have been available by default along with the Python installation (accessible with `python -m venv`).

`venv` allows you to create isolated versions of the Python environment. Using `venv`, you can have many different versions of the same package installed on the same system at the same time without conflicts. This means you can work on many different projects and use many different tools on the same computer. `venv` does this by installing explicit versions of packages and their dependencies into completely separate directory structures. This makes it possible to reproduce a Python environment that you know will work with your code. It’s a reliable way to avoid surprising breakages.

## Using `venv` on the Command Line

Here’s a quick tutorial on how to use `venv` effectively. Before using the tool, it’s important to note the meaning of the `python3` command line on your system. On my computer, `python3` is located in the `/usr/local/bin` directory and evaluates to version 3.8.0 (see [Item 1: “Know Which Version of Python You’re Using”](#)):

```
$ which python3
/usr/local/bin/python3
$ python3 --version
Python 3.8.0
```

To demonstrate the setup of my environment, I can test that running a command to import the `pytz` module doesn't cause an error. This works because I already have the `pytz` package installed as a global module:

```
$ python3 -c 'import pytz'
$
```

Now, I use `venv` to create a new virtual environment called `myproject`. Each virtual environment must live in its own unique directory. The result of the command is a tree of directories and files that are used to manage the virtual environment:

[Click here to view code image](#)

```
$ python3 -m venv myproject
$ cd myproject
$ ls
bin      include      lib      pyvenv.cfg
```

To start using the virtual environment, I use the `source` command from my shell on the `bin/activate` script. `activate` modifies all of my environment variables to match the virtual environment. It also updates my command-line prompt to include the virtual environment name (“`myproject`”) to make it extremely clear what I'm working on:

```
$ source bin/activate
(myproject)$
```

On Windows the same script is available as:

```
C:\> myproject\Scripts\activate.bat
(myproject) C:>
```

Or with PowerShell as:

[Click here to view code image](#)

```
PS C:\> myproject\Scripts\activate.ps1
(myproject) PS C:>
```



After activation, the path to the python3 command-line tool has moved to within the virtual environment directory:

[Click here to view code image](#)

```
(myproject)$ which python3
/tmp/myproject/bin/python3
(myproject)$ ls -l /tmp/myproject/bin/python3
... -> /usr/local/bin/python3.8
```

This ensures that changes to the outside system will not affect the virtual environment. Even if the outer system upgrades its default python3 to version 3.9, my virtual environment will still explicitly point to version 3.8.

The virtual environment I created with venv starts with no packages installed except for pip and setuptools. Trying to use the pytz package that was installed as a global module in the outside system will fail because it's unknown to the virtual environment:

[Click here to view code image](#)

```
(myproject)$ python3 -c 'import pytz'
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ModuleNotFoundError: No module named 'pytz'
```

I can use the pip command-line tool to install the pytz module into my virtual environment:

[Click here to view code image](#)

```
(myproject)$ python3 -m pip install pytz
Collecting pytz
  Downloading ...
Installing collected packages: pytz
Successfully installed pytz-2019.1
```

Once it's installed, I can verify that it's working by using the same test import command:

```
(myproject)$ python3 -c 'import pytz'
(myproject)$
```

When I'm done with a virtual environment and want to go back to my default system, I use the `deactivate` command. This restores my environment to the system defaults, including the location of the `python3` command-line tool:

```
(myproject)$ which python3
/tmp/myproject/bin/python3
(myproject)$ deactivate
$ which python3
/usr/local/bin/python3
```

If I ever want to work in the `myproject` environment again, I can just run `source bin/activate` in the directory as before.

## Reproducing Dependencies

Once you are in a virtual environment, you can continue installing packages in it with `pip` as you need them. Eventually, you might want to copy your environment somewhere else. For example, say that I want to reproduce the development environment from my workstation on a server in a datacenter. Or maybe I want to clone someone else's environment on my own machine so I can help debug their code.

`venv` makes such tasks easy. I can use the `python3 -m pip freeze` command to save all of my explicit package dependencies into a file (which, by convention, is named `requirements.txt`):

[Click here to view code image](#)

```
(myproject)$ python3 -m pip freeze > requirements.txt
(myproject)$ cat requirements.txt
certifi==2019.3.9
chardet==3.0.4
idna==2.8
numpy==1.16.2
pytz==2018.9
requests==2.21.0
urllib3==1.24.1
```

Now, imagine that I'd like to have another virtual environment that matches the `myproject` environment. I can create a new directory as before by using `venv` and activate it:

```
$ python3 -m venv otherproject
$ cd otherproject
$ source bin/activate
(otherproject)$
```

The new environment will have no extra packages installed:

[Click here to view code image](#)

```
(otherproject)$ python3 -m pip list
Package      Version
-----
pip           10.0.1
setuptools   39.0.1
```

I can install all of the packages from the first environment by running `python3 -m pip install` on the `requirements.txt` that I generated with the `python3 -m pip freeze` command:

[Click here to view code image](#)

```
(otherproject)$ python3 -m pip install -r /tmp/myproject/
➔ requirements.txt
```

This command cranks along for a little while as it retrieves and installs all of the packages required to reproduce the first environment. When it's done, I can list the set of installed packages in the second virtual environment and should see the same list of dependencies found in the first virtual environment:

[Click here to view code image](#)

```
(otherproject)$ python3 -m pip list
Package      Version
-----
certifi       2019.3.9
chardet       3.0.4
idna          2.8
numpy         1.16.2
pip           10.0.1
pytz          2018.9
requests      2.21.0
setuptools    39.0.1
urllib3       1.24.1
```

Using a `requirements.txt` file is ideal for collaborating with others through a revision control system. You can commit changes to your code at the same time you update your list of package dependencies, ensuring that they move in lockstep. However, it's important to note that the specific version of Python you're using is *not* included in the `requirements.txt` file, so that must be managed separately.

The gotcha with virtual environments is that moving them breaks everything because all of the paths, like the `python3` command-line tool, are hard-coded to the environment's install directory. But ultimately this limitation doesn't matter. The whole purpose of virtual environments is to make it easy to reproduce a setup. Instead of moving a virtual environment directory, just use `python3 -m pip freeze` on the old one, create a new virtual environment somewhere else, and reinstall everything from the `requirements.txt` file.

## Things to Remember

- ◆ Virtual environments allow you to use `pip` to install many different versions of the same package on the same machine without conflicts.
- ◆ Virtual environments are created with `python -m venv`, enabled with `source bin/activate`, and disabled with `deactivate`.
- ◆ You can dump all of the requirements of an environment with `python3 -m pip freeze`. You can reproduce an environment by running `python3 -m pip install -r requirements.txt`.

## Item 84: Write Docstrings for Every Function, Class, and Module

Documentation in Python is extremely important because of the dynamic nature of the language. Python provides built-in support for attaching documentation to blocks of code. Unlike with many other languages, the documentation from a program's source code is directly accessible as the program runs.

For example, you can add documentation by providing a *docstring* immediately after the `def` statement of a function:

[Click here to view code image](#)

```
def palindrome(word):  
    """Return True if the given word is a palindrome."""  
    return word == word[::-1]  
  
assert palindrome('tacocat')  
assert not palindrome('banana')
```

You can retrieve the docstring from within the Python program by accessing the function's `__doc__` special attribute:

[Click here to view code image](#)

```
print(repr(palindrome.__doc__))  
  
>>>  
'Return True if the given word is a palindrome.'
```

You can also use the built-in `pydoc` module from the command line to run a local web server that hosts all of the Python documentation that's accessible to your interpreter, including modules that you've written:

[Click here to view code image](#)

```
$ python3 -m pydoc -p 1234  
Server ready at http://localhost:1234/  
Server commands: [b]rowser, [q]uit  
server> b
```

Docstrings can be attached to functions, classes, and modules. This connection is part of the process of compiling and running a Python program. Support for docstrings and the `__doc__` attribute has three consequences:

- The accessibility of documentation makes interactive development easier. You can inspect functions, classes, and modules to see their documentation by using the `help` built-in function. This makes the Python interactive interpreter (the Python “shell”) and tools like

IPython Notebook (<https://ipython.org>) a joy to use while you're developing algorithms, testing APIs, and writing code snippets.

- A standard way of defining documentation makes it easy to build tools that convert the text into more appealing formats (like HTML). This has led to excellent documentation-generation tools for the Python community, such as Sphinx (<https://www.sphinx-doc.org>). It has also enabled community-funded sites like Read the Docs (<https://readthedocs.org>) that provide free hosting of beautiful-looking documentation for open source Python projects.
- Python's first-class, accessible, and good-looking documentation encourages people to write more documentation. The members of the Python community have a strong belief in the importance of documentation. There's an assumption that "good code" also means well-documented code. This means that you can expect most open source Python libraries to have decent documentation.

To participate in this excellent culture of documentation, you need to follow a few guidelines when you write docstrings. The full details are discussed online in PEP 257 (<https://www.python.org/dev/peps/pep-0257/>). There are a few best practices you should be sure to follow.

## Documenting Modules

Each module should have a top-level docstring—a string literal that is the first statement in a source file. It should use three double quotes ("""). The goal of this docstring is to introduce the module and its contents.

The first line of the docstring should be a single sentence describing the module's purpose. The paragraphs that follow should contain the details that all users of the module should know about its operation. The module docstring is also a jumping-off point where you can highlight important classes and functions found in the module.

Here's an example of a module docstring:

[Click here to view code image](#)

```
# words.py
#!/usr/bin/env python3
"""Library for finding linguistic patterns in words.

Testing how words relate to each other can be tricky sometimes!
This module provides easy ways to determine when words you've
found have special properties.

Available functions:
- palindrome: Determine if a word is a palindrome.
- check_anagram: Determine if two words are anagrams.
...
"""
...
```

If the module is a command-line utility, the module docstring is also a great place to put usage information for running the tool.

## Documenting Classes

Each class should have a class-level docstring. This largely follows the same pattern as the module-level docstring. The first line is the single-sentence purpose of the class. Paragraphs that follow discuss important details of the class's operation.

Important public attributes and methods of the class should be highlighted in the class-level docstring. It should also provide guidance to subclasses on how to properly interact with protected attributes (see [Item 42: “Prefer Public Attributes Over Private Ones”](#)) and the superclass's methods.

Here's an example of a class docstring:

[Click here to view code image](#)

```
class Player:
    """Represents a player of the game.

    Subclasses may override the 'tick' method to provide
    custom animations for the player's movement depending
    on their power level, etc.

    Public attributes:
    - power: Unused power-ups (float between 0 and 1).
    - coins: Coins found during the level (integer).
    """
```

...

## Documenting Functions

Each public function and method should have a docstring. This follows the same pattern as the docstrings for modules and classes. The first line is a single-sentence description of what the function does. The paragraphs that follow should describe any specific behaviors and the arguments for the function. Any return values should be mentioned. Any exceptions that callers must handle as part of the function's interface should be explained (see [Item 20: “Prefer Raising Exceptions to Returning None”](#) for how to document raised exceptions).

Here's an example of a function docstring:

[Click here to view code image](#)

```
def find_anagrams(word, dictionary):  
    """Find all anagrams for a word.  
  
    This function only runs as fast as the test for  
    membership in the 'dictionary' container.  
  
    Args:  
        word: String of the target word.  
        dictionary: collections.abc.Container with all  
                   strings that are known to be actual words.  
  
    Returns:  
        List of anagrams that were found. Empty if  
        none were found.  
    """  
    ...
```

There are also some special cases in writing docstrings for functions that are important to know:

- If a function has no arguments and a simple return value, a single-sentence description is probably good enough.
- If a function doesn't return anything, it's better to leave out any mention of the return value instead of saying “returns None.”



- If a function’s interface includes raising exceptions (see [Item 20: “Prefer Raising Exceptions to Returning None”](#) for an example), its docstring should describe each exception that’s raised and when it’s raised.
- If you don’t expect a function to raise an exception during normal operation, don’t mention that fact.
- If a function accepts a variable number of arguments (see [Item 22: “Reduce Visual Noise with Variable Positional Arguments”](#)) or keyword arguments (see [Item 23: “Provide Optional Behavior with Keyword Arguments”](#)), use `*args` and `**kwargs` in the documented list of arguments to describe their purpose.
- If a function has arguments with default values, those defaults should be mentioned (see [Item 24: “Use None and Docstrings to Specify Dynamic Default Arguments”](#)).
- If a function is a generator (see [Item 30: “Consider Generators Instead of Returning Lists”](#)), its docstring should describe what the generator yields when it’s iterated.
- If a function is an asynchronous coroutine (see [Item 60: “Achieve Highly Concurrent I/O with Coroutines”](#)), its docstring should explain when it will stop execution.

## Using Docstrings and Type Annotations

Python now supports type annotations for a variety of purposes (see [Item 90: “Consider Static Analysis via typing to Obviate Bugs”](#) for how to use them). The information they contain may be redundant with typical docstrings. For example, here is the function signature for `find_anagrams` with type annotations applied:

[Click here to view code image](#)

```
from typing import Container, List

def find_anagrams(word: str,
                  dictionary: Container[str]) -> List[str]:
    ...
```

There is no longer a need to specify in the docstring that the `word` argument is a string, since the type annotation has that information. The same goes for the dictionary argument being a `collections.abc.Container`. There's no reason to mention that the return type will be a `list`, since this fact is clearly annotated. And when no anagrams are found, the return value still must be a `list`, so it's implied that it will be empty; that doesn't need to be noted in the docstring. Here, I write the same function signature from above along with the docstring that has been shortened accordingly:

[Click here to view code image](#)

```
def find_anagrams(word: str,
                    dictionary: Container[str]) -> List[str]:
    """Find all anagrams for a word.

    This function only runs as fast as the test for
    membership in the 'dictionary' container.

    Args:
        word: Target word.
        dictionary: All known actual words.

    Returns:
        Anagrams that were found.
    """
    ...
```

The redundancy between type annotations and docstrings should be similarly avoided for instance fields, class attributes, and methods. It's best to have type information in only one place so there's less risk that it will skew from the actual implementation.

## Things to Remember

- ✦ Write documentation for every module, class, method, and function using docstrings. Keep them up-to-date as your code changes.
- ✦ For modules: Introduce the contents of a module and any important classes or functions that all users should know about.
- ✦ For classes: Document behavior, important attributes, and subclass behavior in the docstring following the `class` statement.

- ◆ For functions and methods: Document every argument, returned value, raised exception, and other behaviors in the docstring following the `def` statement.
- ◆ If you're using type annotations, omit the information that's already present in type annotations from docstrings since it would be redundant to have it in both places.

## Item 85: Use Packages to Organize Modules and Provide Stable APIs

As the size of a program's codebase grows, it's natural for you to reorganize its structure. You'll split larger functions into smaller functions. You'll refactor data structures into helper classes (see [Item 37: “Compose Classes Instead of Nesting Many Levels of Built-in Types”](#) for an example). You'll separate functionality into various modules that depend on each other.

At some point, you'll find yourself with so many modules that you need another layer in your program to make it understandable. For this purpose, Python provides *packages*. Packages are modules that contain other modules.

In most cases, packages are defined by putting an empty file named `__init__.py` into a directory. Once `__init__.py` is present, any other Python files in that directory will be available for import, using a path relative to the directory. For example, imagine that I have the following directory structure in my program:

```
main.py
mypackage/__init__.py
mypackage/models.py
mypackage/utils.py
```

To import the `utils` module, I use the absolute module name that includes the package directory's name:

```
# main.py
from mypackage import utils
```

This pattern continues when I have package directories present within other packages (like `mypackage.foo.bar`).

The functionality provided by packages has two primary purposes in Python programs.

## Namespaces

The first use of packages is to help divide your modules into separate namespaces. They enable you to have many modules with the same filename but different absolute paths that are unique. For example, here's a program that imports attributes from two modules with the same filename, `utils.py`:

[Click here to view code image](#)

```
# main.py
from analysis.utils import log_base2_bucket
from frontend.utils import stringify
```

```
bucket = stringify(log_base2_bucket(33))
```

This approach breaks when the functions, classes, or submodules defined in packages have the same names. For example, say that I want to use the `inspect` function from both the `analysis.utils` and the `frontend.utils` modules. Importing the attributes directly won't work because the second import statement will overwrite the value of `inspect` in the current scope:

[Click here to view code image](#)

```
# main2.py
from analysis.utils import inspect
from frontend.utils import inspect # Overwrites!
```

The solution is to use the `as` clause of the `import` statement to rename whatever I've imported for the current scope:

[Click here to view code image](#)

```
# main3.py
from analysis.utils import inspect as analysis_inspect
from frontend.utils import inspect as frontend_inspect
```

```
value = 33
if analysis_inspect(value) == frontend_inspect(value):
    print('Inspection equal!')
```

The `as` clause can be used to rename anything retrieved with the `import` statement, including entire modules. This facilitates accessing namespaced code and makes its identity clear when you use it.

Another approach for avoiding imported name conflicts is to always access names by their highest unique module name. For the example above, this means I'd use basic `import` statements instead of `import from`:

[Click here to view code image](#)

```
# main4.py
import analysis.utils
import frontend.utils

value = 33
if (analysis.utils.inspect(value) ==
    frontend.utils.inspect(value)):
    print('Inspection equal!')
```

This approach allows you to avoid the `as` clause altogether. It also makes it abundantly clear to new readers of the code where each of the similarly named functions is defined.

## Stable APIs

The second use of packages in Python is to provide strict, stable APIs for external consumers.

When you're writing an API for wider consumption, such as an open source package (see [Item 82: "Know Where to Find Community-Built Modules"](#) for examples), you'll want to provide stable functionality that doesn't change between releases. To ensure that happens, it's important to hide your internal code organization from external users. This way, you can refactor and improve your package's internal modules without breaking existing users.

Python can limit the surface area exposed to API consumers by using the `__all__` special attribute of a module or package. The value of `__all__` is a

list of every name to export from the module as part of its public API. When consuming code executes from `foo import *`, only the attributes in `foo.__all__` will be imported from `foo`. If `__all__` isn't present in `foo`, then only public attributes—those without a leading underscore—are imported (see [Item 42: “Prefer Public Attributes Over Private Ones”](#) for details about that convention).

For example, say that I want to provide a package for calculating collisions between moving projectiles. Here, I define the `models` module of `mypackage` to contain the representation of projectiles:

[Click here to view code image](#)

```
# models.py
__all__ = ['Projectile']

class Projectile:
    def __init__(self, mass, velocity):
        self.mass = mass
        self.velocity = velocity
```

I also define a `utils` module in `mypackage` to perform operations on the `Projectile` instances, such as simulating collisions between them:

```
# utils.py
from . models import Projectile

__all__ = ['simulate_collision']

def _dot_product(a, b):
    ...

def simulate_collision(a, b):
    ...
```

Now, I'd like to provide all of the public parts of this API as a set of attributes that are available on the `mypackage` module. This will allow downstream consumers to always import directly from `mypackage` instead of importing from `mypackage.models` or `mypackage.utils`. This ensures that the API consumer's code will continue to work even if the internal organization of `mypackage` changes (e.g., `models.py` is deleted).

To do this with Python packages, you need to modify the `__init__.py` file in the `mypackage` directory. This file is what actually becomes the contents of the `mypackage` module when it's imported. Thus, you can specify an explicit API for `mypackage` by limiting what you import into `__init__.py`. Since all of my internal modules already specify `__all__`, I can expose the public interface of `mypackage` by simply importing everything from the internal modules and updating `__all__` accordingly:

```
# __init__.py
__all__ = []
from . models import *

__all__ += models.__all__
from . utils import *
__all__ += utils.__all__
```

Here's a consumer of the API that directly imports from `mypackage` instead of accessing the inner modules:

```
# api_consumer.py
from mypackage import *

a = Projectile(1.5, 3)
b = Projectile(4, 1.7)
after_a,
after_b = simulate_collision(a, b)
```

Notably, internal-only functions like `mypackage.utils._dot_product` will not be available to the API consumer on `mypackage` because they weren't present in `__all__`. Being omitted from `__all__` also means that they weren't imported by the `from mypackage import *` statement. The internal-only names are effectively hidden.

This whole approach works great when it's important to provide an explicit, stable API. However, if you're building an API for use between your own modules, the functionality of `__all__` is probably unnecessary and should be avoided. The namespacing provided by packages is usually enough for a team of programmers to collaborate on large amounts of code they control while maintaining reasonable interface boundaries.

## Beware of `import *`

Import statements like `from x import y` are clear because the source of `y` is explicitly the `x` package or module. Wildcard imports like `from foo import *` can also be useful, especially in interactive Python sessions. However, wildcards make code more difficult to understand:

- `from foo import *` hides the source of names from new readers of the code. If a module has multiple `import *` statements, you'll need to check all of the referenced modules to figure out where a name was defined.
- Names from `import *` statements will overwrite any conflicting names within the containing module. This can lead to strange bugs caused by accidental interactions between your code and overlapping names from multiple `import *` statements.

The safest approach is to avoid `import *` in your code and explicitly import names with the `from x import y` style.

## Things to Remember

- ♦ Packages in Python are modules that contain other modules. Packages allow you to organize your code into separate, non-conflicting namespaces with unique absolute module names.
- ♦ Simple packages are defined by adding an `__init__.py` file to a directory that contains other source files. These files become the child modules of the directory's package. Package directories may also contain other packages.
- ♦ You can provide an explicit API for a module by listing its publicly visible names in its `__all__` special attribute.



- ◆ You can hide a package’s internal implementation by only importing public names in the package’s `__init__.py` file or by naming internal-only members with a leading underscore.
- ◆ When collaborating within a single team or on a single codebase, using `__all__` for explicit APIs is probably unnecessary.

## Item 86: Consider Module-Scoped Code to Configure Deployment Environments

A deployment environment is a configuration in which a program runs. Every program has at least one deployment environment: the *production environment*. The goal of writing a program in the first place is to put it to work in the production environment and achieve some kind of outcome.

Writing or modifying a program requires being able to run it on the computer you use for developing. The configuration of your *development environment* may be very different from that of your production environment. For example, you may be using a tiny single-board computer to develop a program that’s meant to run on enormous supercomputers.

Tools like `venv` (see [Item 83: “Use Virtual Environments for Isolated and Reproducible Dependencies”](#)) make it easy to ensure that all environments have the same Python packages installed. The trouble is that production environments often require many external assumptions that are hard to reproduce in development environments.

For example, say that I want to run a program in a web server container and give it access to a database. Every time I want to modify my program’s code, I need to run a server container, the database schema must be set up properly, and my program needs the password for access. This is a very high cost if all I’m trying to do is verify that a one-line change to my program works correctly.

The best way to work around such issues is to override parts of a program at startup time to provide different functionality depending on the deployment environment. For example, I could have two different `__main__` files—one for production and one for development:

```
# dev_main.py
TESTING = True

import db_connection

db = db_connection.Database()

# prod_main.py
TESTING = False

import db_connection

db = db_connection.Database()
```

The only difference between the two files is the value of the `TESTING` constant. Other modules in my program can then import the `__main__` module and use the value of `TESTING` to decide how they define their own attributes:

```
# db_connection.py
import __main__

class TestingDatabase:
    ...

class RealDatabase:
    ...

if __main__.TESTING:
    Database = TestingDatabase
else:
    Database = RealDatabase
```

The key behavior to notice here is that code running in module scope—not inside a function or method—is just normal Python code. You can use an `if` statement at the module level to decide how the module will define names. This makes it easy to tailor modules to your various deployment environments. You can avoid having to reproduce costly assumptions like database configurations when they aren’t needed. You can inject local or fake implementations that ease interactive development, or you can use mocks for writing tests (see [Item 78: “Use Mocks to Test Code with Complex Dependencies”](#)).

## Note

When your deployment environment configuration gets really complicated, you should consider moving it out of Python constants (like `TESTING`) and into dedicated configuration files. Tools like the `configparser` built-in module let you maintain production configurations separately from code, a distinction that's crucial for collaborating with an operations team.

This approach can be used for more than working around external assumptions. For example, if I know that my program must work differently depending on its host platform, I can inspect the `sys` module before defining top-level constructs in a module:

[Click here to view code image](#)

```
# db_connection.py
import sys

class Win32Database:
    ...

class PosixDatabase:
    ...

if sys.platform.startswith('win32'):
    Database = Win32Database
else:
    Database = PosixDatabase
```

Similarly, I could use environment variables from `os.environ` to guide my module definitions.

## Things to Remember

- ◆ Programs often need to run in multiple deployment environments that each have unique assumptions and configurations.
- ◆ You can tailor a module's contents to different deployment environments by using normal Python statements in module scope.

- ◆ Module contents can be the product of any external condition, including host introspection through the `sys` and `os` modules.

## Item 87: Define a Root `Exception` to Insulate Callers from APIs

When you're defining a module's API, the exceptions you raise are just as much a part of your interface as the functions and classes you define (see [Item 20: "Prefer Raising Exceptions to Returning `None`"](#) for an example).

Python has a built-in hierarchy of exceptions for the language and standard library. There's a draw to using the built-in exception types for reporting errors instead of defining your own new types. For example, I could raise a `ValueError` exception whenever an invalid parameter is passed to a function in one of my modules:

[Click here to view code image](#)

```
# my_module.py
def determine_weight(volume, density):
    if density <= 0:
        raise ValueError('Density must be positive')
    ...
```

In some cases, using `ValueError` makes sense, but for APIs, it's much more powerful to define a new hierarchy of exceptions. I can do this by providing a root `Exception` in my module and having all other exceptions raised by that module inherit from the root exception:

[Click here to view code image](#)

```
# my_module.py
class Error(Exception):
    """Base-class for all exceptions raised by this module."""

class InvalidDensityError(Error):
    """There was a problem with a provided density value."""

class InvalidVolumeError(Error):
    """There was a problem with the provided weight value."""

def determine_weight(volume, density):
```

```
if density < 0:
    raise InvalidDensityError('Density must be positive')
if volume < 0:
    raise InvalidVolumeError('Volume must be positive')
if volume == 0:
    density / volume
```

Having a root exception in a module makes it easy for consumers of an API to catch all of the exceptions that were raised deliberately. For example, here a consumer of my API makes a function call with a try/except statement that catches my root exception:

[Click here to view code image](#)

```
try:
    weight = my_module.determine_weight(1, -1)
except my_module.Error:
    logging.exception('Unexpected error')

>>>
Unexpected error
Traceback (most recent call last):
  File ".../example.py", line 3, in <module>
    weight = my_module.determine_weight(1, -1)
  File ".../my_module.py", line 10, in determine_weight
    raise InvalidDensityError('Density must be positive')
InvalidDensityError: Density must be positive
```

Here, the logging.exception function prints the full stack trace of the caught exception so it's easier to debug in this situation. The try/ except also prevents my API's exceptions from propagating too far upward and breaking the calling program. It insulates the calling code from my API. This insulation has three helpful effects.

First, root exceptions let callers understand when there's a problem with their usage of an API. If callers are using my API properly, they should catch the various exceptions that I deliberately raise. If they don't handle such an exception, it will propagate all the way up to the insulating except block that catches my module's root exception. That block can bring the exception to the attention of the API consumer, providing an opportunity for them to add proper handling of the missed exception type:

[Click here to view code image](#)

```

try:
    weight = my_module.determine_weight(-1, 1)
except my_module.InvalidDensityError:
    weight = 0
except my_module.Error:
    logging.exception('Bug in the calling code')

>>>
Bug in the calling code
Traceback (most recent call last):
  File ".../example.py", line 3, in <module>
    weight = my_module.determine_weight(-1, 1)
  File ".../my_module.py", line 12, in determine_weight
    raise InvalidVolumeError('Volume must be positive')
InvalidVolumeError: Volume must be positive

```

The second advantage of using root exceptions is that they can help find bugs in an API module's code. If my code only deliberately raises exceptions that I define within my module's hierarchy, then all other types of exceptions raised by my module must be the ones that I didn't intend to raise. These are bugs in my API's code.

Using the try/except statement above will not insulate API consumers from bugs in my API module's code. To do that, the caller needs to add another except block that catches Python's base Exception class.

This allows the API consumer to detect when there's a bug in the API module's implementation that needs to be fixed. The output for this example includes both the logging.exception message and the default interpreter output for the exception since it was re-raised:

[Click here to view code image](#)

```

try:
    weight = my_module.determine_weight(0, 1)
except my_module.InvalidDensityError:
    weight = 0
except my_module.Error:
    logging.exception('Bug in the calling code')
except Exception:
    logging.exception('Bug in the API code!')
    raise # Re-raise exception to the caller

>>>

```

```
Bug in the API code!
Traceback (most recent call last):
  File ".../example.py", line 3, in <module>
    weight = my_module.determine_weight(0, 1)
  File ".../my_module.py", line 14, in determine_weight
    density / volume
ZeroDivisionError: division by zero
Traceback ...
ZeroDivisionError: division by zero
```

The third impact of using root exceptions is future-proofing an API. Over time, I might want to expand my API to provide more specific exceptions in certain situations. For example, I could add an `Exception` subclass that indicates the error condition of supplying negative densities:

[Click here to view code image](#)

```
# my_module.py
...

class NegativeDensityError(InvalidDensityError):
    """A provided density value was negative."""
    ...

def determine_weight(volume, density):
    if density < 0:
        raise NegativeDensityError('Density must be positive')
    ...
```

The calling code will continue to work exactly as before because it already catches `InvalidDensityError` exceptions (the parent class of `NegativeDensityError`). In the future, the caller could decide to special-case the new type of exception and change the handling behavior accordingly:

[Click here to view code image](#)

```
try:
    weight = my_module.determine_weight(1, -1)
except my_module.NegativeDensityError:
    raise ValueError('Must supply non-negative density')
except my_module.InvalidDensityError:
    weight = 0
except my_module.Error:
    logging.exception('Bug in the calling code')
```

```
except Exception:
    logging.exception('Bug in the API code!')
    raise
```

```
>>>
```

```
Traceback ...
```

```
NegativeDensityError: Density must be positive
```

The above exception was the direct cause of the following

➔ exception:

```
Traceback ...
```

```
ValueError: Must supply non-negative density
```

I can take API future-proofing further by providing a broader set of exceptions directly below the root exception. For example, imagine that I have one set of errors related to calculating weights, another related to calculating volume, and a third related to calculating density:

[Click here to view code image](#)

```
# my_module.py
class Error(Exception):
    """Base-class for all exceptions raised by this module."""

class WeightError(Error):
    """Base-class for weight calculation errors."""

class VolumeError(Error):
    """Base-class for volume calculation errors."""

class DensityError(Error):
    """Base-class for density calculation errors."""

...
```

Specific exceptions would inherit from these general exceptions. Each intermediate exception acts as its own kind of root exception. This makes it easier to insulate layers of calling code from API code based on broad functionality. This is much better than having all callers catch a long list of very specific Exception subclasses.

## Things to Remember



- ✦ Defining root exceptions for modules allows API consumers to insulate themselves from an API.
- ✦ Catching root exceptions can help you find bugs in code that consumes an API.
- ✦ Catching the Python `Exception` base class can help you find bugs in API implementations.
- ✦ Intermediate root exceptions let you add more specific types of exceptions in the future without breaking your API consumers.

## Item 88: Know How to Break Circular Dependencies

Inevitably, while you're collaborating with others, you'll find a mutual interdependence between modules. It can even happen while you work by yourself on the various parts of a single program.

For example, say that I want my GUI application to show a dialog box for choosing where to save a document. The data displayed by the dialog could be specified through arguments to my event handlers. But the dialog also needs to read global state, such as user preferences, to know how to render properly.

Here, I define a dialog that retrieves the default document save location from global preferences:

[Click here to view code image](#)

```
# dialog.py
import app

class Dialog:
    def __init__(self, save_dir):
        self.save_dir = save_dir

    ...

save_dialog = Dialog(app.prefs.get('save_dir'))

def show():
    ...
```

The problem is that the app module that contains the prefs object also imports the dialog class in order to show the same dialog on program start:

```
# app.py
import dialog

class Prefs:
    ...
    def get(self, name):
        ...

prefs = Prefs()
dialog.show()
```

It's a circular dependency. If I try to import the app module from my main program like this:

[Click here to view code image](#)

```
# main.py
import app
```

I get an exception:

```
>>>
$ python3 main.py
Traceback (most recent call last):
  File ".../main.py", line 17, in <module>
    import app
  File ".../app.py", line 17, in <module>
    import dialog
  File ".../dialog.py", line 23, in <module>
    save_dialog = Dialog(app.prefs.get('save_dir'))
AttributeError: partially initialized module 'app' has no
➔ attribute 'prefs' (most likely due to a circular import)
```

To understand what's happening here, you need to know how Python's import machinery works in general (see the `importlib` built-in package for the full details). When a module is imported, here's what Python actually does, in depth-first order:

1. Searches for a module in locations from `sys.path`
2. Loads the code from the module and ensures that it compiles

3. Creates a corresponding empty module object
4. Inserts the module into `sys.modules`
5. Runs the code in the module object to define its contents

The problem with a circular dependency is that the attributes of a module aren't defined until the code for those attributes has executed (after step 5). But the module can be loaded with the `import` statement immediately after it's inserted into `sys.modules` (after step 4).

In the example above, the `app` module imports `dialog` before defining anything. Then, the `dialog` module imports `app`. Since `app` still hasn't finished running—it's currently importing `dialog`—the `app` module is empty (from step 4). The `AttributeError` is raised (during step 5 for `dialog`) because the code that defines `prefs` hasn't run yet (step 5 for `app` isn't complete).

The best solution to this problem is to refactor the code so that the `prefs` data structure is at the bottom of the dependency tree. Then, both `app` and `dialog` can import the same utility module and avoid any circular dependencies. But such a clear division isn't always possible or could require too much refactoring to be worth the effort.

There are three other ways to break circular dependencies.

## Reordering Imports

The first approach is to change the order of imports. For example, if I import the `dialog` module toward the bottom of the `app` module, after the `app` module's other contents have run, the `AttributeError` goes away:

```
# app.py
class Prefs:
    ...

prefs = Prefs()

import dialog # Moved
dialog.show()
```

This works because, when the `dialog` module is loaded late, its recursive import of `app` finds that `app.prefs` has already been defined (step 5 is mostly done for `app`).

Although this avoids the `AttributeError`, it goes against the PEP 8 style guide (see [Item 2: “Follow the PEP 8 Style Guide”](#)). The style guide suggests that you always put imports at the top of your Python files. This makes your module’s dependencies clear to new readers of the code. It also ensures that any module you depend on is in scope and available to all the code in your module.

Having imports later in a file can be brittle and can cause small changes in the ordering of your code to break the module entirely. I suggest not using import reordering to solve your circular dependency issues.

## Import, Configure, Run

A second solution to the circular imports problem is to have modules minimize side effects at import time. I can have my modules only define functions, classes, and constants. I avoid actually running any functions at import time. Then, I have each module provide a `configure` function that I call once all other modules have finished importing. The purpose of `configure` is to prepare each module’s state by accessing the attributes of other modules. I run `configure` after all modules have been imported (step 5 is complete), so all attributes must be defined.

Here, I redefine the `dialog` module to only access the `prefs` object when `configure` is called:

[Click here to view code image](#)

```
# dialog.py
import app

class Dialog:
    ...

save_dialog = Dialog()

def show():
    ...
```

```
def configure():
    save_dialog.save_dir = app.prefs.get('save_dir')
```

I also redefine the app module to not run activities on import:

```
# app.py
import dialog

class Prefs:
    ...

prefs = Prefs()

def configure():
    ...
```

Finally, the main module has three distinct phases of execution— import everything, configure everything, and run the first activity:

```
# main.py
import app
import dialog

app.configure()
dialog.configure()

dialog.show()
```

This works well in many situations and enables patterns like *dependency injection*. But sometimes it can be difficult to structure your code so that an explicit configure step is possible. Having two distinct phases within a module can also make your code harder to read because it separates the definition of objects from their configuration.

## Dynamic Import

The third—and often simplest—solution to the circular imports problem is to use an `import` statement within a function or method. This is called a *dynamic import* because the module import happens while the program is running, not while the program is first starting up and initializing its modules.

Here, I redefine the `dialog` module to use a dynamic import. The `dialog.show` function imports the `app` module at runtime instead of the `dialog` module importing `app` at initialization time:

[Click here to view code image](#)

```
# dialog.py
class Dialog:
    ...

save_dialog = Dialog()

def show():
    import app # Dynamic import
    save_dialog.save_dir = app.prefs.get('save_dir')
    ...
```

The `app` module can now be the same as it was in the original example. It imports `dialog` at the top and calls `dialog.show` at the bottom:

```
# app.py
import dialog
class Prefs:
    ...

prefs = Prefs()
dialog.show()
```

This approach has a similar effect to the `import`, `configure`, and `run` steps from before. The difference is that it requires no structural changes to the way the modules are defined and imported. I’m simply delaying the circular import until the moment I must access the other module. At that point, I can be pretty sure that all other modules have already been initialized (step 5 is complete for everything).

In general, it’s good to avoid dynamic imports like this. The cost of the `import` statement is not negligible and can be especially bad in tight loops. By delaying execution, dynamic imports also set you up for surprising failures at runtime, such as `SyntaxError` exceptions long after your program has started running (see [Item 76: “Verify Related Behaviors in TestCase Subclasses”](#) for how to avoid that). However, these downsides are often better than the alternative of restructuring your entire program.

## Things to Remember

- ✦ Circular dependencies happen when two modules must call into each other at import time. They can cause your program to crash at startup.
- ✦ The best way to break a circular dependency is by refactoring mutual dependencies into a separate module at the bottom of the dependency tree.
- ✦ Dynamic imports are the simplest solution for breaking a circular dependency between modules while minimizing refactoring and complexity.

## Item 89: Consider `warnings` to Refactor and Migrate Usage

It's natural for APIs to change in order to satisfy new requirements that meet formerly unanticipated needs. When an API is small and has few upstream or downstream dependencies, making such changes is straightforward. One programmer can often update a small API and all of its callers in a single commit.

However, as a codebase grows, the number of callers of an API can be so large or fragmented across source repositories that it's infeasible or impractical to make API changes in lockstep with updating callers to match. Instead, you need a way to notify and encourage the people that you collaborate with to refactor their code and migrate their API usage to the latest forms.

For example, say that I want to provide a module for calculating how far a car will travel at a given average speed and duration. Here, I define such a function and assume that speed is in miles per hour and duration is in hours:

[Click here to view code image](#)

```
def print_distance(speed, duration):  
    distance = speed * duration  
    print(f'{distance} miles')  
  
print_distance(5, 2.5)
```

```
>>>
12.5 miles
```

Imagine that this works so well that I quickly gather a large number of dependencies on this function. Other programmers that I collaborate with need to calculate and print distances like this all across our shared codebase.

Despite its success, this implementation is error prone because the units for the arguments are implicit. For example, if I wanted to see how far a bullet travels in 3 seconds at 1000 meters per second, I would get the wrong result:

```
print_distance(1000, 3)
```

```
>>>
3000 miles
```

I can address this problem by expanding the API of `print_distance` to include optional keyword arguments (see [Item 23: “Provide Optional Behavior with Keyword Arguments”](#) and [Item 25: “Enforce Clarity with Keyword-Only and Positional-Only Arguments”](#)) for the units of speed, duration, and the computed distance to print out:

[Click here to view code image](#)

```
CONVERSIONS = {
    'mph': 1.60934 / 3600 * 1000,    # m/s
    'hours': 3600,                  # seconds
    'miles': 1.60934 * 1000,        # m
    'meters': 1,                    # m
    'm/s': 1,                       # m
    'seconds': 1,                   # s
}

def convert(value, units):
    rate = CONVERSIONS[units]
    return rate * value

def localize(value, units):
    rate = CONVERSIONS[units]
    return value / rate

def print_distance(speed, duration, *,
                   speed_units='mph',
                   time_units='hours',
```



```
        distance_units='miles'):
    norm_speed = convert(speed, speed_units)
    norm_duration = convert(duration, time_units)
    norm_distance = norm_speed * norm_duration
    distance = localize(norm_distance, distance_units)
    print(f'{distance} {distance_units}')
```

Now, I can modify the speeding bullet call to produce an accurate result with a unit conversion to miles:

[Click here to view code image](#)

```
print_distance(1000, 3,
               speed_units='meters',
               time_units='seconds')
```

```
>>>
1.8641182099494205 miles
```

It seems like requiring units to be specified for this function is a much better way to go. Making them explicit reduces the likelihood of errors and is easier for new readers of the code to understand. But how can I migrate all callers of the API over to always specifying units? How do I minimize breakage of any code that's dependent on `print_distance` while also encouraging callers to adopt the new units arguments as soon as possible?

For this purpose, Python provides the built-in `warnings` module. Using `warnings` is a programmatic way to inform other programmers that their code needs to be modified due to a change to an underlying library that they depend on. While exceptions are primarily for automated error handling by machines (see [Item 87: “Define a Root Exception to Insulate Callers from APIs”](#)), warnings are all about communication between humans about what to expect in their collaboration with each other.

I can modify `print_distance` to issue warnings when the optional keyword arguments for specifying units are not supplied. This way, the arguments can continue being optional temporarily (see [Item 24: “Use None and Docstrings to Specify Dynamic Default Arguments”](#) for background), while providing an explicit notice to people running dependent programs that they should expect breakage in the future if they fail to take action:

[Click here to view code image](#)

```
import warnings

def print_distance(speed, duration, *,
                  speed_units=None,
                  time_units=None,
                  distance_units=None):
    if speed_units is None:
        warnings.warn(
            'speed_units required', DeprecationWarning)
        speed_units = 'mph'

    if time_units is None:
        warnings.warn(
            'time_units required', DeprecationWarning)
        time_units = 'hours'

    if distance_units is None:
        warnings.warn(
            'distance_units required', DeprecationWarning)
        distance_units = 'miles'

    norm_speed = convert(speed, speed_units)
    norm_duration = convert(duration, time_units)
    norm_distance = norm_speed * norm_duration
    distance = localize(norm_distance, distance_units)
    print(f'{distance} {distance_units}')
```

I can verify that this code issues a warning by calling the function with the same arguments as before and capturing the `sys.stderr` output from the warnings module:

[Click here to view code image](#)

```
import contextlib
import io

fake_stderr = io.StringIO()
with contextlib.redirect_stderr(fake_stderr):
    print_distance(1000, 3,
                  speed_units='meters',
                  time_units='seconds')

print(fake_stderr.getvalue())

>>>
```

```
1.8641182099494205 miles
.../example.py:97: DeprecationWarning: distance_units required
  warnings.warn(
```

Adding warnings to this function required quite a lot of repetitive boilerplate that's hard to read and maintain. Also, the warning message indicates the line where `warnings.warn` was called, but what I really want to point out is where the call to `print_distance` was made *without* soon-to-be-required keyword arguments.

Luckily, the `warnings.warn` function supports the `stacklevel` parameter, which makes it possible to report the correct place in the stack as the cause of the warning. `stacklevel` also makes it easy to write functions that can issue warnings on behalf of other code, reducing boilerplate. Here, I define a helper function that warns if an optional argument wasn't supplied and then provides a default value for it:

[Click here to view code image](#)

```
def require(name, value, default):
    if value is not None:
        return value
    warnings.warn(
        f'{name} will be required soon, update your code',
        DeprecationWarning,
        stacklevel=3)
    return default

def print_distance(speed, duration, *,
                  speed_units=None,
                  time_units=None,
                  distance_units=None):
    speed_units = require('speed_units', speed_units, 'mph')
    time_units = require('time_units', time_units, 'hours')
    distance_units = require(
        'distance_units', distance_units, 'miles')

    norm_speed = convert(speed, speed_units)
    norm_duration = convert(duration, time_units)
    norm_distance = norm_speed * norm_duration
    distance = localize(norm_distance, distance_units)
    print(f'{distance} {distance_units}')
```

I can verify that this propagates the proper offending line by inspecting the captured output:

[Click here to view code image](#)

```
import contextlib
import io

fake_stderr = io.StringIO()
with contextlib.redirect_stderr(fake_stderr):
    print_distance(1000, 3,
                  speed_units='meters',
                  time_units='seconds')

print(fake_stderr.getvalue())

>>>
1.8641182099494205 miles
.../example.py:174: DeprecationWarning: distance_units will be
➔ required soon, update your code
print_distance(1000, 3,
```

The warnings module also lets me configure what should happen when a warning is encountered. One option is to make all warnings become errors, which raises the warning as an exception instead of printing it out to `sys.stderr`:

[Click here to view code image](#)

```
warnings.simplefilter('error')
try:
    warnings.warn('This usage is deprecated',
                  DeprecationWarning)
except DeprecationWarning:
    pass # Expected
```

This exception-raising behavior is especially useful for automated tests in order to detect changes in upstream dependencies and fail tests accordingly. Using such test failures is a great way to make it clear to the people you collaborate with that they will need to update their code. You can use the `-w` error command-line argument to the Python interpreter or the `PYTHONWARNINGS` environment variable to apply this policy:

[Click here to view code image](#)

```
$ python -W error example_test.py
Traceback (most recent call last):
  File ".../example_test.py", line 6, in <module>
    warnings.warn('This might raise an exception!')
UserWarning: This might raise an exception!
```

Once the people responsible for code that depends on a deprecated API are aware that they'll need to do a migration, they can tell the warnings module to ignore the error by using the `simplefilter` and `filterwarnings` functions (see <https://docs.python.org/3/library/warnings> for all the details):

[Click here to view code image](#)

```
warnings.simplefilter('ignore')
warnings.warn('This will not be printed to stderr')
```

After a program is deployed into production, it doesn't make sense for warnings to cause errors because they might crash the program at a critical time. Instead, a better approach is to replicate warnings into the logging built-in module. Here, I accomplish this by calling the `logging.captureWarnings` function and configuring the corresponding 'py.warnings' logger:

[Click here to view code image](#)

```
import logging

fake_stderr = io.StringIO()
handler = logging.StreamHandler(fake_stderr)
formatter = logging.Formatter(
    '%(asctime)s-15s WARNING] %(message)s')
handler.setFormatter(formatter)

logging.captureWarnings(True)
logger = logging.getLogger('py.warnings')
logger.addHandler(handler)
logger.setLevel(logging.DEBUG)

warnings.resetwarnings()
warnings.simplefilter('default')
warnings.warn('This will go to the logs output')

print(fake_stderr.getvalue())

>>>
```

```
2019-06-11 19:48:19,132 WARNING] .../example.py:227:
```

```
➔ UserWarning: This will go to the logs output
warnings.warn('This will go to the logs output')
```

Using logging to capture warnings ensures that any error reporting systems that my program already has in place will also receive notice of important warnings in production. This can be especially useful if my tests don't cover every edge case that I might see when the program is undergoing real usage.

API library maintainers should also write unit tests to verify that warnings are generated under the correct circumstances with clear and actionable messages (see [Item 76: “Verify Related Behaviors in TestCase Subclasses”](#)). Here, I use the `warnings.catch_warnings` function as a context manager (see [Item 66: “Consider contextlib and with Statements for Reusable try/finally Behavior”](#) for background) to wrap a call to the `require` function that I defined above:

[Click here to view code image](#)

```
with warnings.catch_warnings(record=True) as found_warnings:
    found = require('my_arg', None, 'fake units')
    expected = 'fake units'
    assert found == expected
```

Once I've collected the warning messages, I can verify that their number, detail messages, and categories match my expectations:

[Click here to view code image](#)

```
assert len(found_warnings) == 1
single_warning = found_warnings[0]
assert str(single_warning.message) == (
    'my_arg will be required soon, update your code')
assert single_warning.category == DeprecationWarning
```

## Things to Remember

- ◆ The `warnings` module can be used to notify callers of your API about deprecated usage. Warning messages encourage such callers to fix their code before later changes break their programs.

- ◆ Raise warnings as errors by using the `-W error` command-line argument to the Python interpreter. This is especially useful in automated tests to catch potential regressions of dependencies.
- ◆ In production, you can replicate warnings into the logging module to ensure that your existing error reporting systems will capture warnings at runtime.
- ◆ It's useful to write tests for the warnings that your code generates to make sure that they'll be triggered at the right time in any of your downstream dependencies.

## Item 90: Consider Static Analysis via `typing` to Obviate Bugs

Providing documentation is a great way to help users of an API understand how to use it properly (see [Item 84: “Write Docstrings for Every Function, Class, and Module”](#)), but often it's not enough, and incorrect usage still causes bugs. Ideally, there would be a programmatic mechanism to verify that callers are using your APIs the right way, and that you are using your downstream dependencies correctly. Many programming languages address part of this need with compile-time type checking, which can identify and eliminate some categories of bugs.

Historically Python has focused on dynamic features and has not provided compile-time type safety of any kind. However, more recently Python has introduced special syntax and the built-in `typing` module, which allow you to annotate variables, class fields, functions, and methods with type information. These *type hints* allow for *gradual typing*, where a codebase can be incrementally updated to specify types as desired.

The benefit of adding type information to a Python program is that you can run *static analysis* tools to ingest a program's source code and identify where bugs are most likely to occur. The `typing` built-in module doesn't actually implement any of the type checking functionality itself. It merely provides a common library for defining types, including generics, that can be applied to Python code and consumed by separate tools.

Much as there are multiple distinct implementations of the Python interpreter (e.g., CPython, PyPy), there are multiple implementations of static analysis tools for Python that use typing. As of the time of this writing, the most popular tools are mypy (<https://github.com/python/mypy>), pytype (<https://github.com/google/pytype>), pyright (<https://github.com/microsoft/pyright>), and pyre (<https://pyre-check.org>). For the typing examples in this book, I've used mypy with the `--strict` flag, which enables all of the various warnings supported by the tool. Here's an example of what running the command line looks like:

```
$ python3 -m mypy --strict example.py
```

These tools can be used to detect a large number of common errors before a program is ever run, which can provide an added layer of safety in addition to having good unit tests (see [Item 76: “Verify Related Behaviors in TestCase Subclasses”](#)). For example, can you find the bug in this simple function that causes it to compile fine but throw an exception at runtime?

[Click here to view code image](#)

```
def subtract(a, b):  
    return a - b
```

```
subtract(10, '5')
```

```
>>>
```

```
Traceback ...
```

```
TypeError: unsupported operand type(s) for -: 'int' and 'str'
```

Parameter and variable type annotations are delineated with a colon (such as `name: type`). Return value types are specified with `-> type` following the argument list. Using such type annotations and mypy, I can easily spot the bug:

[Click here to view code image](#)

```
def subtract(a: int, b: int) -> int: # Function annotation  
    return a - b
```

```
subtract(10, '5') # Oops: passed string value
```

```
$ python3 -m mypy --strict example.py
```



```
.../example.py:4: error: Argument 2 to "subtract" has
incompatible type "str"; expected "int"
```

Another common mistake, especially for programmers who have recently moved from Python 2 to Python 3, is mixing bytes and str instances together (see [Item 3: “Know the Differences Between bytes and str”](#)). Do you see the problem in this example that causes a runtime error?

[Click here to view code image](#)

```
def concat(a, b):
    return a + b

concat('first', b'second')

>>>
Traceback ...
TypeError: can only concatenate str (not "bytes") to str
```

Using type hints and mypy, this issue can be detected statically before the program runs:

[Click here to view code image](#)

```
def concat(a: str, b: str) -> str:
    return a + b

concat('first', b'second') # Oops: passed bytes value

$ python3 -m mypy --strict example.py
.../example.py:4: error: Argument 2 to "concat" has
→ incompatible type "bytes"; expected "str"
```

Type annotations can also be applied to classes. For example, this class has two bugs in it that will raise exceptions when the program is run:

```
class Counter:
    def __init__(self):
        self.value = 0

    def add(self, offset):
        value += offset

    def get(self) -> int:
        self.value
```

The first one happens when I call the add method:

[Click here to view code image](#)

```
counter = Counter()
counter.add(5)

>>>
Traceback ...
UnboundLocalError: local variable 'value' referenced before
➔ assignment
```

The second bug happens when I call get:

```
counter = Counter()
found = counter.get()
assert found == 0, found

>>>
Traceback ...
AssertionError: None
```

Both of these problems are easily found by mypy:

[Click here to view code image](#)

```
class Counter:
    def __init__(self) -> None:
        self.value: int = 0 # Field / variable annotation

    def add(self, offset: int) -> None:
        value += offset      # Oops: forgot "self."

    def get(self) -> int:
        self.value           # Oops: forgot "return"

counter = Counter()
counter.add(5)
counter.add(3)
assert counter.get() == 8

$ python3 -m mypy --strict example.py
.../example.py:6: error: Name 'value' is not defined
.../example.py:8: error: Missing return statement
```

One of the strengths of Python's dynamism is the ability to write generic functionality that operates on duck types (see [Item 15](#): “Be Cautious When

[Relying on dict Insertion Ordering](#)” and [Item 43: “Inherit from collections.abc for Custom Container Types”](#)). This allows one implementation to accept a wide range of types, saving a lot of duplicative effort and simplifying testing. Here, I’ve defined such a generic function for combining values from a list. Do you understand why the last assertion fails?

```
def combine(func, values):
    assert len(values) > 0

    result = values[0]
    for next_value in values[1:]:
        result = func(result, next_value)

    return result

def add(x, y):
    return x + y

inputs = [1, 2, 3, 4j]
result = combine(add, inputs)
assert result == 10, result # Fails
```

```
>>>
Traceback ...
AssertionError: (6+4j)
```

I can use the typing module’s support for generics to annotate this function and detect the problem statically:

[Click here to view code image](#)

```
from typing import Callable, List, TypeVar

Value = TypeVar('Value')
Func = Callable[[Value, Value], Value]

def combine(func: Func[Value], values: List[Value]) -> Value:
    assert len(values) > 0

    result = values[0]
    for next_value in values[1:]:
        result = func(result, next_value)

    return result
```

```

Real = TypeVar('Real', int, float)

def add(x: Real, y: Real) -> Real:
    return x + y

inputs = [1, 2, 3, 4j] # Oops: included a complex number
result = combine(add, inputs)
assert result == 10

$ python3 -m mypy --strict example.py
.../example.py:21: error: Argument 1 to "combine" has
➔ incompatible type "Callable[[Real, Real], Real]"; expected
➔ "Callable[[complex, complex], complex]"

```

Another extremely common error is to encounter a `None` value when you thought you'd have a valid object (see [Item 20: “Prefer Raising Exceptions to Returning None”](#)). This problem can affect seemingly simple code. Do you see the issue here?

```

def get_or_default(value, default):
    if value is not None:
        return value
    return value

found = get_or_default(3, 5)
assert found == 3

found = get_or_default(None, 5)
assert found == 5, found # Fails

>>>
Traceback ...
AssertionError: None

```

The typing module supports *option types*, which ensure that programs only interact with values after proper null checks have been performed. This allows mypy to infer that there's a bug in this code: The type used in the return statement must be `None`, and that doesn't match the `int` type required by the function signature:

[Click here to view code image](#)

```

from typing import Optional

```

```
def get_or_default(value: Optional[int],
                  default: int) -> int:
    if value is not None:
        return value
    return value # Oops: should have returned "default"
```

```
$ python3 -m mypy --strict example.py
.../example.py:7: error: Incompatible return value type (got
➔ "None", expected "int")
```

A wide variety of other options are available in the typing module. See <https://docs.python.org/3.8/library/typing> for all of the details. Notably, exceptions are not included. Unlike Java, which has checked exceptions that are enforced at the API boundary of every method, Python's type annotations are more similar to C#'s: Exceptions are not considered part of an interface's definition. Thus, if you want to verify that you're raising and catching exceptions properly, you need to write tests.

One common gotcha in using the typing module occurs when you need to deal with forward references (see [Item 88: “Know How to Break Circular Dependencies”](#) for a similar problem). For example, imagine that I have two classes and one holds a reference to the other:

```
class FirstClass:
    def __init__(self, value):
        self.value = value

class SecondClass:
    def __init__(self, value):
        self.value = value
```

```
second = SecondClass(5)
first = FirstClass(second)
```

If I apply type hints to this program and run mypy it will say that there are no issues:

[Click here to view code image](#)

```
class FirstClass:
    def __init__(self, value: SecondClass) -> None:
        self.value = value

class SecondClass:
```

```
def __init__(self, value: int) -> None:
    self.value = value
```

```
second = SecondClass(5)
first = FirstClass(second)
```

```
$ python3 -m mypy --strict example.py
```

However, if you actually try to run this code, it will fail because `SecondClass` is referenced by the type annotation in the `FirstClass.__init__` method's parameters before it's actually defined:

[Click here to view code image](#)

```
class FirstClass:
    def __init__(self, value: SecondClass) -> None: # Breaks
        self.value = value
```

```
class SecondClass:
    def __init__(self, value: int) -> None:
        self.value = value
```

```
second = SecondClass(5)
first = FirstClass(second)
```

```
>>>
```

```
Traceback ...
```

```
NameError: name 'SecondClass' is not defined
```

One workaround supported by these static analysis tools is to use a string as the type annotation that contains the forward reference. The string value is later parsed and evaluated to extract the type information to check:

[Click here to view code image](#)

```
class FirstClass:
    def __init__(self, value: 'SecondClass') -> None: # OK
        self.value = value
```

```
class SecondClass:
    def __init__(self, value: int) -> None:
        self.value = value
```

```
second = SecondClass(5)
first = FirstClass(second)
```

A better approach is to use `from __future__ import annotations`, which is available in Python 3.7 and will become the default in Python 4. This instructs the Python interpreter to completely ignore the values supplied in type annotations when the program is being run. This resolves the forward reference problem and provides a performance improvement at program start time:

[Click here to view code image](#)

```
from __future__ import annotations

class FirstClass:
    def __init__(self, value: SecondClass) -> None: # OK
        self.value = value

class SecondClass:
    def __init__(self, value: int) -> None:
        self.value = value

second = SecondClass(5)
first = FirstClass(second)
```

Now that you've seen how to use type hints and their potential benefits, it's important to be thoughtful about when to use them. Here are some of the best practices to keep in mind:

- It's going to slow you down if you try to use type annotations from the start when writing a new piece of code. A general strategy is to write a first version without annotations, then write tests, and then add type information where it's most valuable.
- Type hints are most important at the boundaries of a codebase, such as an API you provide that many callers (and thus other people) depend on. Type hints complement integration tests (see [Item 77: “Isolate Tests from Each Other with `setUp`, `tearDown`, `setUpModule`, and `tearDownModule`”](#)) and warnings (see [Item 89: “Consider warnings to Refactor and Migrate Usage”](#)) to ensure that your API callers aren't surprised or broken by your changes.
- It can be useful to apply type hints to the most complex and errorprone parts of your codebase that aren't part of an API. However, it may not

be worth striving for 100% coverage in your type annotations because you'll quickly encounter diminishing returns.

- If possible, you should include static analysis as part of your automated build and test system to ensure that every commit to your codebase is vetted for errors. In addition, the configuration used for type checking should be maintained in the repository to ensure that all of the people you collaborate with are using the same rules.
- As you add type information to your code, it's important to run the type checker as you go. Otherwise, you may nearly finish sprinkling type hints everywhere and then be hit by a huge wall of errors from the type checking tool, which can be disheartening and make you want to abandon type hints altogether.

Finally, it's important to acknowledge that in many situations, you may not need or want to use type annotations at all. For small programs, ad-hoc code, legacy codebases, and prototypes, type hints may require far more effort than they're worth.

## Things to Remember

- ♦ Python has special syntax and the `typing` built-in module for annotating variables, fields, functions, and methods with type information.
- ♦ Static type checkers can leverage type information to help you avoid many common bugs that would otherwise happen at runtime.
- ♦ There are a variety of best practices for adopting types in your programs, using them in APIs, and making sure they don't get in the way of your productivity.



# Index

## Symbols

\* (asterisk) operator

keyword-only arguments, [98](#)

variable positional arguments, [87–88](#)

@ (at) symbol, decorators, [101](#)

\*\* (double asterisk) operator, keyword arguments, [90–91](#)

/ (forward slash) operator, positional-only arguments, [99](#)

% (percent) operator

bytes versus str instances, [8–9](#)

formatting strings, [11](#)

+ (plus) operator, bytes versus str instances, [7](#)

\_ (underscore) variable name, [149](#)

:= (walrus) operator

assignment expression, [35–41](#)

in comprehensions, [112–114](#)

\_\_call\_\_ method, [154–155](#)

@classmethod, [155–160](#)

\_\_format\_\_ method, [16](#)

\_\_getattr\_\_ method, [195–201](#)

\_\_getattribute\_\_ method, [195–201](#)

\_\_init\_\_ method, [160–164](#)

\_\_init\_subclass\_\_ method

registering classes, [208–213](#)

validating subclasses, [201–208](#)

\_\_iter\_\_ method, [119](#), [244–245](#)

\_\_missing\_\_ method (dictionary subclasses), [73–75](#)

@property decorator

descriptors versus, [190–195](#)

- refactoring attributes with, [186–189](#)
- setter attribute, [182–185](#)
- `__set_name__` method, annotating attributes, [214–218](#)
- `__setattr__` method, [195–201](#)
- CPython, [230](#)

## A

### APIs

- migrating usage, [418–425](#)
- root exceptions for, [408–413](#)
- stability, [403–405](#)

### arguments

- dynamic default values, [93–96](#)
- iterating over, [116–121](#)
- keyword, [89–92](#)
- keyword-only, [96–101](#)
- positional-only, [96–101](#)
- variable positional, [86–89](#)

assertions in TestCase subclasses, [359](#)

### assignment expressions

- in comprehensions, [110–114](#)
- scope and, [85](#)
- walrus (`:=`) operator, [35–41](#)

associative arrays, [43](#)

### asterisk (\*) operator

- keyword-only arguments, [98](#)
- variable positional arguments, [87–88](#)

### asyncio built-in module

- avoiding blocking, [289–292](#)
- combining threads and coroutines, [282–288](#)
- porting threaded I/O to, [271–282](#)

at (@) symbol, decorators, [101](#)

attributes

- annotating, [214–218](#)

- dynamic, [181](#)

- getter and setter methods versus, [181–185](#)

- lazy, [195–201](#)

- public versus private, [169–174](#)

- refactoring, [186–189](#)

## B

binary data, converting to Unicode, [6–7](#)

binary operators, bytes versus str instances, [8](#)

bisect built-in module, [334–336](#)

blocking asyncio event loop, avoiding, [289–292](#)

blocking I/O (input/output) with threads, [230–235](#)

breaking circular dependencies, [413–418](#)

breakpoint built-in function, [379–384](#)

buffer protocol, [348](#)

built-in types, classes versus, [145–148](#)

bytearray built-in type, [346–351](#)

bytecode, [230](#)

bytes instances, str instances versus, [5–10](#)

## C

C extensions, [292–293](#)

C3 linearization, [162](#)

callables, [154](#)

catch-all unpacking, slicing versus, [48–52](#)

character data, bytes versus str instances, [5–10](#)

checked exceptions, [82](#)

child processes, managing, [226–230](#)

circular dependencies, breaking, [413–418](#)

classes, [145](#)

- attributes. *See* [attributes](#)

- built-in types versus, [145–148](#)

- decorators, [218–224](#)

- documentation, [398–399](#)

- function interfaces versus, [151–155](#)

- initializing parent classes, [160–164](#)

- metaclasses. *See* [metaclasses](#)

- mix-in classes, [164–169](#)

- polymorphism, [155–160](#)

- public versus private attributes, [169–174](#)

- refactoring to, [148–151](#)

- registering, [208–213](#)

- serializing, [168–169](#)

- validating subclasses, [201–208](#)

- versioning, [316–317](#)

closures, variable scope and, [83–86](#)

collaboration

- breaking circular dependencies, [413–418](#)

  - dynamic import, [417–418](#)

  - import/configure/run, [415–416](#)

  - reordering imports, [415–416](#)

- community-built modules, [389–390](#)

- documentation, [396–401](#)

- migrating API usage, [418–425](#)

- organizing modules into packages, [401–406](#)

- root exceptions for APIs, [408–413](#)

- static analysis, [425–434](#)

- virtual environments, [390–396](#)

collections.abc module, inheritance from, [174–178](#)

combining iterator items, [139–142](#)

- commands for interactive debugger, [381](#)
- community-built modules, [389–390](#)
- compile-time static type checking, [353](#)
- complex sort criteria with key parameter, [52–58](#)
- comprehensions, [107](#)
  - assignment expressions in, [110–114](#)
  - generator expressions for, [121–122](#)
  - map and filter functions versus, [107–109](#)
  - multiple subexpressions in, [109–110](#)
- concurrency, [225](#)
  - avoiding threads for fan-out, [252–256](#)
  - fan-in, [252](#)
  - fan-out, [252](#)
  - highly concurrent I/O (input/output), [266–271](#)
  - parallelism versus, [225](#)
  - with pipelines, [238–247](#)
  - preventing data races, [235–238](#)
  - using Queue class for, [257–263](#)
  - using ThreadPoolExecutor for, [264–266](#)
  - with threads, [230–235](#)
  - when to use, [248–252](#)
- concurrent.futures built-in module, [292–297](#)
- configuring deployment environments, [406–408](#)
- conflicts with dependencies, [390–396](#)
- containers
  - inheritance from collections.abc module, [174–178](#)
  - iterator protocol, [119–121](#)
- contextlib built-in module, [304–308](#)
- Coordinated Universal Time (UTC), [308](#)
- copyreg built-in module, [312–319](#)
- coroutines, [266–271](#)
  - combining with threads, [282–288](#)

C-style strings, f-strings versus, [11–21](#)

custom container types, inheritance from collections.abc module, [174–178](#)

## D

data races, preventing, [235–238](#)

datetime built-in module, [308–312](#)

debugging

- with interactive debugger, [379–384](#)

- memory usage, [384–387](#)

- with repr strings, [354–357](#)

- with static analysis, [425–434](#)

Decimal class, rounding numbers, [319–322](#)

decorators

- class decorators, [218–224](#)

- function decorators, [101–104](#)

default arguments

- dynamic, [93–96](#)

- with pickle built-in module, [315–316](#)

default values in dictionaries

- `__missing__` method, [73–75](#)

- `defaultdict` versus `setdefault` methods, [70–72](#)

- `get` method versus in expressions, [65–70](#)

`defaultdict` class, `setdefault` method versus, [70–72](#)

dependencies

- breaking circular, [413–418](#)

- conflicts, [390–396](#)

- encapsulating, [375–379](#)

- injecting, [378–379](#)

- reproducing, [394–396](#)

- testing with mocks, [367–375](#)

dependency hell, [391](#)

- deployment environments, configuring, [406–408](#)
- deque class, [326–334](#)
- descriptor protocol, [191](#)
- descriptors versus @property decorator, [190–195](#)
- deserializing with pickle built-in module, [312–319](#)
- development environment, [406–407](#)
- diamond inheritance, [161–162](#), [207–208](#)
- dictionaries, [43](#)
  - insertion ordering, [58–65](#)
  - missing keys
    - \_\_missing\_\_ method, [73–75](#)
    - defaultdict versus setdefault methods, [70–72](#)
    - get method versus in expressions, [65–70](#)
  - nesting, [145–148](#)
  - tuples versus in format strings, [13–15](#)
- dictionary comprehensions, [108–109](#)
- docstrings
  - for dynamic default arguments, [93–96](#)
  - writing, [396–401](#)
    - for classes, [398–399](#)
    - for functions, [399–400](#)
    - for modules, [397–398](#)
    - type annotations and, [400–401](#)
  - documentation. See [docstrings](#)
- double asterisk (\*\*) operator, keyword arguments, [90–91](#)
- double-ended queues, [331](#)
- duck typing, [61](#), [429](#)
- dynamic attributes, [181](#)
- dynamic default arguments, [93–96](#)
- dynamic import, [417–418](#)

## E

else blocks

- for statements, [32–35](#)

- exception handling, [299–304](#)

encapsulating dependencies, [375–379](#)

enumerate built-in function, range built-in function versus, [28–30](#)

except blocks, exception handling, [299–304](#)

exception handling with try/except/else/finally blocks, [299–304](#)

exceptions

- raising, None return value versus, [80–82](#)

- root exceptions for APIs, [408–413](#)

expressions

- helper functions versus, [21–24](#)

- PEP 8 style guide, [4](#)

## F

fakes, mocks versus, [368](#)

fan-in, [252](#)

- with Queue class, [257–263](#)

- with ThreadPoolExecutor class, [264–265](#)

fan-out, [252](#)

- avoiding threads for, [252–256](#)

- with Queue class, [257–263](#)

- with ThreadPoolExecutor class, [264–265](#)

FIFO (first-in, first-out) queues, [326–334](#)

file operations, bytes versus str instances, [9–10](#)

filter built-in function, comprehensions versus, [107–109](#)

finally blocks

- exception handling, [299–304](#)

- with statements versus, [304–308](#)

first-class functions, [152](#)



- first-in, first-out (FIFO) queues, [326–334](#)
- for loops, avoiding else blocks, [32–35](#)
- format built-in function, [15–19](#)
- format strings
  - bytes versus str instances, [8–9](#)
  - C-style strings versus f-strings, [11–21](#)
  - format built-in function, [15–19](#)
  - f-strings explained, [19–21](#)
  - interpolated format strings, [19–21](#)
  - problems with C-style strings, [11–15](#)
  - str.format method, [15–19](#)
- forward slash (/) operator, positional-only arguments, [99](#)
- f-strings
  - C-style strings versus, [11–21](#)
  - str.format method versus, [15–19](#)
  - explained, [19–21](#)
- functions, [77](#). *See also* [generators](#)
  - closures, variable scope and, [83–86](#)
  - decorators, [101–104](#)
  - documentation, [399–400](#)
  - dynamic default arguments, [93–96](#)
  - as hooks, [151–155](#)
  - keyword arguments, [89–92](#)
  - keyword-only arguments, [96–101](#)
  - None return value, raising exceptions versus, [80–82](#)
  - in pipelines, [238–247](#)
  - positional-only arguments, [96–101](#)
  - multiple return values, [77–80](#)
  - variable positional arguments, [86–89](#)
- functools.wraps method, [101–104](#)

## G

- gc built-in module, [384–386](#)
- generator expressions, [121–122](#)
- generators, [107](#)
  - yield from for composing, [123–126](#)
  - injecting data into, [126–131](#)
  - itertools module with, [136–142](#)
  - returning lists versus, [114–116](#)
  - send method, [126–131](#)
  - throw method, [132–136](#)
- generic object construction, [155–160](#)
- get method for missing dictionary keys, [65–70](#)
- getter methods, attributes versus, [181–185](#)
- GIL (global interpreter lock), [230–235](#), [292](#)
- gradual typing, [426](#)

## H

- hasattr built-in function, [198–199](#)
- hash tables, [43](#)
- heapq built-in module, [336–346](#)
- heaps, [341](#)
- helper functions, expressions versus, [21–24](#)
- highly concurrent I/O, [266–271](#)
- hooks, functions as, [151–155](#)

## I

- if/else conditional expressions, [23](#)
- import paths, stabilizing, [317–319](#)
- importing modules, [5](#), [414–415](#)
- in expressions for missing dictionary keys, [65–70](#)
- indexing
  - slicing and, [44](#)

unpacking versus, [24–28](#)

inheritance

from collections.abc module, [174–178](#)

diamond inheritance, [161–162](#), [207–208](#)

initializing parent classes, [160–164](#)

injecting

data into generators, [126–131](#)

dependencies, [378–379](#)

mocks, [371–375](#)

input/output (I/O). *See* [I/O \(input/output\)](#)

insertion ordering, dictionaries, [58–65](#)

installing modules, [389–390](#)

integration tests, unit tests versus, [365](#)

interactive debugging, [379–384](#)

interfaces, [145](#)

simple functions for, [151–155](#)

interpolated format strings. *See* [f-strings](#)

I/O (input/output)

avoiding blocking asyncio event loop, [289–292](#)

using threads for, [230–235](#)

highly concurrent, [266–271](#)

porting threaded I/O to asyncio built-in module, [271–282](#)

zero-copy interactions, [346–351](#)

isolating tests, [365–367](#)

iterator protocol, [119–121](#)

iterators. *See also* [loops](#)

combining items, [139–142](#)

filtering items, [138–139](#)

generator expressions and, [121–122](#)

generator functions and, [115–116](#)

linking, [136–138](#)

- as function arguments, [116–121](#)
- StopIteration exception, [117](#)
- itertools module, [136–142](#)
- itertools.accumulate method, [139–140](#)
- itertools.chain method, [136](#)
- itertools.combinations method, [141](#)
- itertools.combinations\_with\_replacement method, [141–142](#)
- itertools.cycle method, [137](#)
- itertools.dropwhile method, [139](#)
- itertools.filterfalse method, [139](#)
- itertools.islice method, [138](#)
- itertools.permutations method, [140–141](#)
- itertools.product method, [140](#)
- itertools.repeat method, [136](#)
- itertools.takewhile method, [138](#)
- itertools.tee method, [137](#)
- itertools.zip\_longest method, [31–32](#), [137–138](#)

## J

- json built-in module, [313](#)

## K

- key parameter, sorting lists, [52–58](#)
- KeyError exceptions for missing dictionary keys, [65–70](#)
- keys
  - handling in dictionaries
    - \_\_missing\_\_ method, [73–75](#)
    - defaultdict versus setdefault methods, [70–72](#)
    - get method versus in expressions, [65–70](#)
- keyword arguments, [89–92](#)
- keyword-only arguments, [96–101](#)

## L

lazy attributes, [195–201](#)

leaks (memory), debugging, [384–387](#)

linking iterators, [136–138](#)

list comprehensions, [107–108](#)

    generator expressions versus, [121–122](#)

lists, [43](#). *See also* [comprehensions](#)

    as FIFO queues, [326–331](#)

    as return values, generators versus, [114–116](#)

    slicing, [43–46](#)

        catch-all unpacking versus, [48–52](#)

        striding with, [46–48](#)

    sorting

        with key parameter, [52–58](#)

        searching sorted lists, [334–336](#)

local time, [308–312](#)

Lock class, preventing data races, [235–238](#)

loops. *See also* [comprehensions](#)

    else blocks, avoiding, [32–35](#)

    range versus enumerate built-in functions, [28–30](#)

    zip built-in function, [30–32](#)

## M

map built-in function, comprehensions versus, [107–109](#)

memory usage, debugging, [384–387](#)

memoryview built-in type, [346–351](#)

metaclasses, [181](#)

    annotating attributes, [214–218](#)

    class decorators versus, [218–224](#)

    registering classes, [208–213](#)

    validating subclasses, [201–208](#)

- migrating API usage, [418–425](#)
- missing dictionary keys
  - `__missing__` method, [73–75](#)
  - `defaultdict` versus `setdefault` methods, [70–72](#)
  - `get` method versus in expressions, [65–70](#)
- mix-in classes, [164–169](#)
- mocks
  - encapsulating dependencies for, [375–379](#)
  - testing with, [367–375](#)
- modules
  - documentation, [397–398](#)
  - importing, [5](#), [414–415](#)
    - dynamic import, [417–418](#)
    - `import/configure/run`, [415–416](#)
    - reordering imports, [415–416](#)
  - installing, [389–390](#)
  - organizing into packages, [401–406](#)
- module-scoped code, [406–408](#)
- multiple assignment. *See* [tuples](#)
- multiple return values, unpacking, [77–80](#)
- multiple generators, composing with `yield` from expression, [123–126](#)
- multiprocessing built-in module, [292–297](#)
- multi-threaded program, converting from single-threaded to, [248–252](#)
- mutexes (mutual-exclusion locks), preventing data races, [235–238](#)

## N

- `namedtuple` type, [149–150](#)
- namespaces, [402–403](#)
- naming conventions, [3–4](#)
- negative numbers for slicing, [44](#)
- nested built-in types, classes versus, [145–148](#)

None

- for dynamic default arguments, [93–96](#)

- raising exceptions versus returning, [80–82](#)

nonlocal statement, [85–86](#)

## O

objects, generic construction, [155–160](#)

optimizing, profiling before, [322–326](#)

option types, [430](#)

optional arguments, extending functions with, [92](#)

OrderedDict class, [61](#)

organizing modules into packages, [401–406](#)

## P

packages

- installing, [389–390](#)

- organizing modules into, [401–406](#)

parallel iteration, zip built-in function, [30–32](#)

parallelism, [225](#)

- avoiding threads, [230–235](#)

- concurrency versus, [225](#)

- with concurrent.futures built-in module, [292–297](#)

- managing child processes, [226–230](#)

parent classes, initializing, [160–164](#)

pdb built-in module, [379–384](#)

PEP 8 style guide, [2–5](#)

percent (%) operator

- bytes versus str instances, [8–9](#)

- dictionaries versus tuples with, [13–15](#)

- formatting strings, [11](#)

performance, [299](#)

- first-in, first-out (FIFO) queues, [326–334](#)
- priority queues, [336–346](#)
- profiling before optimizing, [322–326](#)
- searching sorted lists, [334–336](#)
- zero-copy interactions, [346–351](#)
- pickle built-in module, [312–319](#)
- pip command-line tool, [389–390](#)
- pipelines
  - coordinating threads with, [238–247](#)
  - parallel processes, chains of, [228–229](#)
  - refactoring to use Queue for, [257–263](#)
- plus (+) operator, bytes versus str instances, [7](#)
- polymorphism, [155–160](#)
- porting threaded I/O to asyncio built-in module, [271–282](#)
- positional arguments, variable, [86–89](#)
- positional-only arguments, [96–101](#)
- post-mortem debugging, [382–384](#)
- print function, debugging with, [354–357](#)
- priority queues, [336–346](#)
- private attributes, public attributes versus, [169–174](#)
- processes, managing child processes, [226–230](#)
- ProcessPoolExecutor class, [295–297](#)
- producer-consumer queues, [326–334](#)
- production environment, [406](#)
- profiling before optimizing, [322–326](#)
- public attributes, private attributes versus, [169–174](#)
- Pylint, [5](#)
- PyPI (Python Package Index), [389–390](#)
- Python
  - determining version used, [1–2](#)
  - style guide. See [PEP 8 style guide](#)



Python 2, [1–2](#)

Python 3, [1–2](#)

Python Enhancement Proposal #8. *See* [PEP 8 style guide](#)

Python Package Index (PyPI), [389–390](#)

Pythonic style, [1](#)

pytz module, [311–312](#)

## Q

Queue class

- coordinating threads with, [238–247](#)

- refactoring to use for concurrency, [257–263](#)

## R

raising exceptions, None return value versus, [80–82](#)

range built-in function, enumerate built-in function versus, [28–30](#)

refactoring

- attributes, [186–189](#)

- to break circular dependencies, [415](#)

- to classes, [148–151](#)

- to use Queue class for concurrency, [257–263](#)

registering classes, [208–213](#)

reordering imports, [415–416](#)

repetitive code, avoiding, [35–41](#)

repr strings, debugging with, [354–357](#)

reproducing dependencies, [394–396](#)

return values

- generators versus lists as, [114–116](#)

- None return value, raising exceptions versus, [80–82](#)

- unpacking multiple, [77–80](#)

reusable @property methods, [190–195](#)

reusable try/finally blocks, [304–308](#)

- robustness, [299](#)
  - exception handling with try/except/else/finally blocks, [299–304](#)
  - reusable try/finally blocks, [304–308](#)
  - rounding numbers, [319–322](#)
  - serialization/deserialization with pickle, [312–319](#)
  - time zone conversion, [308–312](#)
- root exceptions for APIs, [408–413](#)
- rounding numbers with Decimal class, [319–322](#)
- rule of least surprise, [181](#)

## S

- scope, closures and, [83–86](#)
- scoping bug, [85](#)
- searching sorted lists, [334–336](#)
- send method in generators, [126–131](#)
- sequences
  - searching sorted, [334–336](#)
  - slicing, [43–46](#)
    - catch-all unpacking versus, [48–52](#)
    - striding, [46–48](#)
- serializing
  - classes, [168–169](#)
  - with pickle built-in module, [312–319](#)
- set comprehensions, [108–109](#)
- setdefault method (dictionaries), [68–70](#)
  - defaultdict method versus, [70–72](#)
- setter methods, attributes versus, [181–185](#)
- setUp method (TestCase class), [365–367](#)
- setUpModule function, [365–367](#)
- single-threaded program, converting to multi-threaded, [248–252](#)
- slicing

- memoryview instances, [348](#)
- sequences, [43–46](#)
  - catch-all unpacking versus, [48–52](#)
  - striding, [46–48](#)
- software licensing, [390](#)
- sorting
  - dictionaries, insertion ordering, [58–65](#)
  - lists
    - with key parameter, [52–58](#)
    - searching sorted lists, [334–336](#)
- speedup, [225](#)
- stabilizing import paths, [317–319](#)
- stable APIs, [403–405](#)
- stable sorting, [56–57](#)
- star args, [86–89](#)
- starred expressions, [49–52](#)
- statements, PEP 8 style guide, [4](#)
- static analysis, [425–434](#)
- StopIteration exception, [117](#)
- str instances, bytes instances versus, [5–10](#)
- str.format method, [15–19](#)
- striding, [46–48](#)
- strings, C-style versus f-strings, [11–21](#)
  - format built-in function, [15–19](#)
  - interpolated format strings, [19–21](#)
  - problems with C-style strings, [11–15](#)
  - str.format method, [15–19](#)
- subclasses, validating, [201–208](#)
- subexpressions in comprehensions, [109–110](#)
- subprocess built-in module, [226–230](#)
- super built-in function, [160–164](#)

## T

tearDown method (TestCase class), [365–367](#)

tearDownModule function, [365–367](#)

ternary expressions, [23](#)

test harness, [365](#)

TestCase subclasses

- isolating tests, [365–367](#)

- verifying related behaviors, [357–365](#)

testing

- encapsulating dependencies for, [375–379](#)

- importance of, [353–354](#)

- isolating tests, [365–367](#)

- with mocks, [367–375](#)

- with TestCase subclasses, [357–365](#)

- unit versus integration tests, [365](#)

- with unittest built-in module, [357](#)

ThreadPoolExecutor class, [264–266](#)

threads

- avoiding for fan-out, [252–256](#)

- combining with coroutines, [282–288](#)

- converting from single- to multi-threaded program, [248–252](#)

- coordinating between, [238–247](#)

- porting threaded I/O to asyncio built-in module, [271–282](#)

- preventing data races, [235–238](#)

- refactoring to use Queue class for concurrency, [257–263](#)

- ThreadPoolExecutor class, [264–266](#)

- when to use, [230–235](#)

throw method in generators, [132–136](#)

time built-in module, [308–312](#)

time zone conversion, [308–312](#)

timeout parameter for subprocesses, [229–230](#)

tracemalloc built-in module, [384–387](#)

try blocks

- exception handling, [299–304](#)

- versus with statements, [304–308](#)

tuples

- dictionaries versus with format strings, [13–15](#)

- indexing versus unpacking, [24–28](#)

- namedtuple type, [149–150](#)

- sorting with multiple criteria, [55–56](#)

- underscore (`_`) variable name in, [149](#)

type annotations, [82](#)

- docstrings and, [400–401](#)

- with static analysis, [425–434](#)

type hints, [426](#)

typing built-in module, [425–434](#)

## U

underscore (`_`) variable name, [149](#)

Unicode data, converting to binary, [6–7](#)

unit tests, integration tests versus, [365](#)

unittest built-in module, [357](#)

unpacking

- indexing versus, [24–28](#)

- multiple return values, [77–80](#)

- slicing versus, [48–52](#)

UTC (Coordinated Universal Time), [308](#)

## V

validating subclasses, [201–208](#)

variable positional arguments (varargs), [86–89](#)

variable scope, closures and, [83–86](#)

- venv built-in module, [392–394](#)
- versioning classes, [316–317](#)
- versions of Python, determining version used, [1–2](#)
- virtual environments, [390–396](#)

## W

- walrus (:=) operator
  - assignment expression, [35–41](#)
  - in comprehensions, [112–114](#)
- warnings built-in module, [418–425](#)
- weakref built-in module, [194](#)
- while loops, avoiding else blocks, [32–35](#)
- whitespace, [3](#)
- with statements for reusable try/finally blocks, [304–308](#)
- with as targets, [306–308](#)
- writing docstrings, [396–401](#)
  - for classes, [398–399](#)
  - for functions, [399–400](#)
  - for modules, [397–398](#)
  - type annotations and, [400–401](#)

## Y

- yield from expressions, composing multiple generators, [123–126](#)

## Z

- zero-copy interactions, [346–351](#)
- zip built-in function, [30–32](#)



*Effective Python LiveLessons* provides visual demonstration of the items presented in the book version so you can see each item in action. Watch Brett Slatkin provide concise and specific guidance on what to do and what to avoid when writing programs using Python.

**Watch and learn how to:**

- Use expressions and statements more efficiently
- Make better use of comprehensions and generators
- Work with concurrency and parallelism
- Make better use of functions and classes
- Make your programs more robust

Save 60%\*—Use coupon code VIDEOBOB  
**[informit.com/slatkin/video](http://informit.com/slatkin/video)**

\*Discount code VIDEOBOB confers a 60% discount off the list price of featured video when purchased on InformIT. Offer is subject to change.





Photo by izusek/gettyimages

## Register Your Product at [informit.com/register](https://informit.com/register)

Access additional benefits and **save 35%** on your next purchase

- Automatically receive a coupon for 35% off your next purchase, valid for 30 days. Look for your code in your InformIT cart or the Manage Codes section of your account page.
- Download available product updates.
- Access bonus material if available.\*
- Check the box to hear from us and receive exclusive offers on new editions and related products.

*\*Registration benefits vary by product. Benefits will be listed on your account page under Registered Products.*

---

### InformIT.com—The Trusted Technology Learning Source

InformIT is the online home of information technology brands at Pearson, the world's foremost education company. At InformIT.com, you can:

- Shop our books, eBooks, software, and video training
- Take advantage of our special offers and promotions ([informit.com/promotions](https://informit.com/promotions))
- Sign up for special offers and content newsletter ([informit.com/newsletters](https://informit.com/newsletters))
- Access thousands of free chapters and video lessons

Connect with InformIT—Visit [informit.com/community](https://informit.com/community)



**informIT**<sup>®</sup>  
the trusted technology learning source

Addison-Wesley • Adobe Press • Cisco Press • Microsoft Press • Pearson IT Certification • Prentice Hall • Que • Sams • Peachpit Press





## Code Snippets

Many titles include programming code or configuration examples. To optimize the presentation of these elements, view the eBook in single-column, landscape mode and adjust the font size to the smallest setting. In addition to presenting code and configurations in the reflowable text format, we have included images of the code that mimic the presentation found in the print book; therefore, where the reflowable format may compromise the presentation of the code listing, you will see a “Click here to view code image” link. Click the link to view the print-fidelity code image. To return to the previous page viewed, click the Back button on your device or app.

```
import sys
print(sys.version_info)
print(sys.version)

>>>
sys.version_info(major=3, minor=8, micro=0,
➡releaselevel='final', serial=0)
3.8.0 (default, Oct 21 2019, 12:51:32)
[Clang 6.0 (clang-600.0.57)]
```

```
a = 'a\u0300 propos'
```

```
print(list(a))
```

```
print(a)
```

```
>>>
```

```
['a', ' ', ' ', 'p', 'r', 'o', 'p', 'o', 's']
```

```
à propos
```

```
def to_str(bytes_or_str):
    if isinstance(bytes_or_str, bytes):
        value = bytes_or_str.decode('utf-8')
    else:
        value = bytes_or_str
    return value # Instance of str
print(repr(to_str(b'foo')))
print(repr(to_str('bar')))

>>>
'foo'
'bar'
```

```
def to_bytes(bytes_or_str):
    if isinstance(bytes_or_str, str):
        value = bytes_or_str.encode('utf-8')
    else:
        value = bytes_or_str
    return value # Instance of bytes

print(repr(to_bytes(b'foo')))
print(repr(to_bytes('bar')))
```

```
b'one' + 'two'
```

```
>>>
```

```
Traceback ...
```

```
TypeError: can't concat str to bytes
```

```
'one' + b'two'
```

```
>>>
```

```
Traceback ...
```

```
TypeError: can only concatenate str (not "bytes") to str
```

```
assert 'red' > b'blue'
```

```
>>>
```

```
Traceback ...
```

```
TypeError: '>' not supported between instances of 'str' and  
↳ 'bytes'
```



```
assert b'blue' < 'red'
```

```
>>>
```

```
Traceback ...
```

```
TypeError: '<' not supported between instances of 'bytes'  
↳and 'str'
```

```
print(b'red %s' % 'blue')
```

```
>>>
```

```
Traceback ...
```

```
TypeError: %b requires a bytes-like object, or an object that  
↳ implements __bytes__, not 'str'
```

```
with open('data.bin', 'w') as f:  
    f.write(b'\xf1\xf2\xf3\xf4\xf5')
```

```
>>>
```

```
Traceback ...
```

```
TypeError: write() argument must be str, not bytes
```

```
with open('data.bin', 'r') as f:  
    data = f.read()
```

```
>>>
```

```
Traceback ...
```

```
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xf1 in  
➡position 0: invalid continuation byte
```

```
with open('data.bin', 'rb') as f:  
    data = f.read()  
  
assert data == b'\xf1\xf2\xf3\xf4\xf5'
```

```
with open('data.bin', 'r', encoding='cp1252') as f:  
    data = f.read()  
  
assert data == 'ñòóôõ'
```

```
a = 0b10111011
b = 0xc5f
print('Binary is %d, hex is %d' % (a, b))

>>>
Binary is 187, hex is 3167
```

```
key = 'my_var'  
value = 1.234  
formatted = '%-10s = %.2f' % (key, value)  
print(formatted)
```

```
>>>
```

```
my_var      = 1.23
```



```
reordered_tuple = '%-10s = %.2f' % (value, key)
```

```
>>>
```

```
Traceback ...
```

```
TypeError: must be real number, not str
```

```
reordered_string = '%.2f = %-10s' % (key, value)
```

```
>>>
```

```
Traceback ...
```

```
TypeError: must be real number, not str
```

```
pantry = [  
    ('avocados', 1.25),  
    ('bananas', 2.5),  
    ('cherries', 15),  
]  
for i, (item, count) in enumerate(pantry):  
    print('#%d: %-10s = %.2f' % (i, item, count))  
  
>>>  
#0: avocados    = 1.25  
#1: bananas     = 2.50  
#2: cherries    = 15.00
```

```
for i, (item, count) in enumerate(pantry):  
    print('#%d: %-10s = %d' % (  
        i + 1,  
        item.title(),  
        round(count)))
```

```
>>>
```

```
#1: Avocados    = 1  
#2: Bananas     = 2  
#3: Cherries    = 15
```

```
template = '%s loves food. See %s cook.'  
name = 'Max'  
formatted = template % (name, name)  
print(formatted)
```

```
>>>
```

```
Max loves food. See Max cook.
```

```
name = 'brad'
formatted = template % (name.title(), name.title())
print(formatted)

>>>
Brad loves food. See Brad cook.
```

```
key = 'my_var'
value = 1.234

old_way = '%-10s = %.2f' % (key, value)

new_way = '%(key)-10s = %(value).2f' % {
    'key': key, 'value': value} # Original

reordered = '%(key)-10s = %(value).2f' % {
    'value': value, 'key': key} # Swapped

assert old_way == new_way == reordered
```

```
name = 'Max'
```

```
template = '%s loves food. See %s cook.'
```

```
before = template % (name, name)    # Tuple
```

```
template = '%(name)s loves food. See %(name)s cook.'
```

```
after = template % {'name': name}   # Dictionary
```

```
assert before == after
```



```
for i, (item, count) in enumerate(pantry):
    before = '#%d: %-10s = %d' % (
        i + 1,
        item.title(),
        round(count))

    after = '#%(loop)d: %(item)-10s = %(count)d' % {
        'loop': i + 1,
        'item': item.title(),
        'count': round(count),
    }

    assert before == after
```

```
soup = 'lentil'
formatted = 'Today\'s soup is %(soup)s.' % {'soup': soup}
print(formatted)

>>>
Today's soup is lentil.
```

```
menu = {
    'soup': 'lentil',
    'oyster': 'kumamoto',
    'special': 'schnitzel',
}
template = ('Today\'s soup is %(soup)s, '
            'buy one get two %(oyster)s oysters, '
            'and our special entrée is %(special)s.')
formatted = template % menu
print(formatted)

>>>
Today's soup is lentil, buy one get two kumamoto oysters, and
our special entrée is schnitzel.
```

```
key = 'my_var'  
value = 1.234
```

```
formatted = '{} = {}'.format(key, value)  
print(formatted)
```

```
>>>  
my_var = 1.234
```

```
formatted = '{:<10} = {:.2f}'.format(key, value)
print(formatted)
```

```
>>>
```

```
my_var      = 1.23
```

```
print('%.2f%%' % 12.5)
print('{} replaces {}'.format(1.23))
```

```
>>>
```

```
12.50%
```

```
1.23 replaces {}
```

```
formatted = '{1} = {0}'.format(key, value)
print(formatted)
```

```
>>>
```

```
1.234 = my_var
```

```
formatted = '{0} loves food. See {0} cook.'.format(name)
print(formatted)
```

```
>>>
```

```
Max loves food. See Max cook.
```



```
for i, (item, count) in enumerate(pantry):
    old_style = '#%d: %-10s = %d' % (
        i + 1,
        item.title(),
        round(count))

    new_style = '#{ }: {:<10s} = {}'.format(
        i + 1,
        item.title(),
        round(count))

    assert old_style == new_style
```

```
formatted = 'First letter is {menu[oyster][0]!r}'.format(  
    menu=menu)  
print(formatted)
```

```
>>>
```

```
First letter is 'k'
```

```
old_template = (  
    'Today\'s soup is %(soup)s, '  
    'buy one get two %(oyster)s oysters, '  
    'and our special entrée is %(special)s.')
```

```
old_formatted = template % {  
    'soup': 'lentil',  
    'oyster': 'kumamoto',  
    'special': 'schnitzel',  
}
```

```
new_template = (  
    'Today\'s soup is {soup}, '  
    'buy one get two {oyster} oysters, '  
    'and our special entrée is {special}.')
```

```
new_formatted = new_template.format(  
    soup='lentil',  
    oyster='kumamoto',  
    special='schnitzel',  
)
```

```
assert old_formatted == new_formatted
```

```
formatted = f'{key!r:<10} = {value:.2f}'  
print(formatted)
```

```
>>>
```

```
'my_var'    = 1.23
```

```
f_string = f'{key:<10} = {value:.2f}'

c_tuple  = '%-10s = %.2f' % (key, value)

str_args = '{:<10} = {:.2f}'.format(key, value)

str_kw   = '{key:<10} = {value:.2f}'.format(key=key,
                                           value=value)

c_dict   = '%(key)-10s = %(value).2f' % {'key': key,
                                         'value': value}

assert c_tuple == c_dict == f_string
assert str_args == str_kw == f_string
```

```
for i, (item, count) in enumerate(pantry):
    old_style = '#%d: %-10s = %d' % (
        i + 1,
        item.title(),
        round(count))

    new_style = '#{i}: {:<10s} = {}'.format(
        i + 1,
        item.title(),
        round(count))

    f_string = f'#{i+1}: {item.title():<10s} = {round(count)}'

    assert old_style == new_style == f_string
```

```
for i, (item, count) in enumerate(pantry):  
    print(f'#{i+1}: '  
          f'{item.title():<10s} = '  
          f'{round(count)}')
```

```
>>>
```

```
#1: Avocados    = 1
```

```
#2: Bananas     = 2
```

```
#3: Cherries    = 15
```

```
places = 3
number = 1.23456
print(f'My number is {number:.{places}f}')

>>>
My number is 1.235
```



```
from urllib.parse import parse_qs
my_values = parse_qs('red=5&blue=0&green=',
                     keep_blank_values=True)
print(repr(my_values))

>>>
{'red': ['5'], 'blue': ['0'], 'green': ['']}
```

```
print('Red:      ', my_values.get('red'))  
print('Green:    ', my_values.get('green'))  
print('Opacity: ', my_values.get('opacity'))
```

```
>>>
```

```
Red:      ['5']
```

```
Green:    ['']
```

```
Opacity:  None
```

```
# For query string 'red=5&blue=0&green='
red = my_values.get('red', [''])[0] or 0
green = my_values.get('green', [''])[0] or 0
opacity = my_values.get('opacity', [''])[0] or 0
print(f'Red:      {red!r}')
print(f'Green:    {green!r}')
print(f'Opacity:  {opacity!r}')
```

```
>>>
```

```
Red:      '5'
```

```
Green:    0
```

```
Opacity:  0
```

```
red = int(my_values.get('red', [' '])[0] or 0)
```

```
red_str = my_values.get('red', [''])  
red = int(red_str[0]) if red_str[0] else 0
```

```
green_str = my_values.get('green', [''])  
if green_str[0]:  
    green = int(green_str[0])  
else:  
    green = 0
```

```
def get_first_int(values, key, default=0):  
    found = values.get(key, [''])  
    if found[0]:  
        return int(found[0])  
    return default
```

```
green = get_first_int(my_values, 'green')
```



```
snack_calories = {  
    'chips': 140,  
    'popcorn': 80,  
    'nuts': 190,  
}  
items = tuple(snack_calories.items())  
print(items)  
  
>>>  
(('chips', 140), ('popcorn', 80), ('nuts', 190))
```

```
pair = ('Chocolate', 'Peanut butter')  
pair[0] = 'Honey'
```

```
>>>
```

```
Traceback ...
```

```
TypeError: 'tuple' object does not support item assignment
```

```
item = ('Peanut butter', 'Jelly')  
first, second = item # Unpacking  
print(first, 'and', second)
```

```
>>>
```

```
Peanut butter and Jelly
```

```
favorite_snacks = {
    'salty': ('pretzels', 100),
    'sweet': ('cookies', 180),
    'veggie': ('carrots', 20),
}

((type1, (name1, cals1)),
 (type2, (name2, cals2)),
 (type3, (name3, cals3))) = favorite_snacks.items()

print(f'Favorite {type1} is {name1} with {cals1} calories')
print(f'Favorite {type2} is {name2} with {cals2} calories')
print(f'Favorite {type3} is {name3} with {cals3} calories')

>>>
Favorite salty is pretzels with 100 calories
Favorite sweet is cookies with 180 calories
Favorite veggie is carrots with 20 calories
```

```
def bubble_sort(a):
    for _ in range(len(a)):
        for i in range(1, len(a)):
            if a[i] < a[i-1]:
                temp = a[i]
                a[i] = a[i-1]
                a[i-1] = temp

names = ['pretzels', 'carrots', 'arugula', 'bacon']
bubble_sort(names)
print(names)

>>>
['arugula', 'bacon', 'carrots', 'pretzels']
```

```
def bubble_sort(a):
    for _ in range(len(a)):
        for i in range(1, len(a)):
            if a[i] < a[i-1]:
                a[i-1], a[i] = a[i], a[i-1] # Swap

names = ['pretzels', 'carrots', 'arugula', 'bacon']
bubble_sort(names)
print(names)

>>>
['arugula', 'bacon', 'carrots', 'pretzels']
```

```
snacks = [('bacon', 350), ('donut', 240), ('muffin', 190)]
for i in range(len(snacks)):
    item = snacks[i]
    name = item[0]
    calories = item[1]
    print(f'#{i+1}: {name} has {calories} calories')

>>>
#1: bacon has 350 calories
#2: donut has 240 calories
#3: muffin has 190 calories
```

```
for rank, (name, calories) in enumerate(snacks, 1):  
    print(f'#{rank}: {name} has {calories} calories')
```

```
>>>
```

```
#1: bacon has 350 calories  
#2: donut has 240 calories  
#3: muffin has 190 calories
```



```
from random import randint

random_bits = 0
for i in range(32):
    if randint(0, 1):
        random_bits |= 1 << i

print(bin(random_bits))

>>>
0b11101000100100000111000010000001
```

```
flavor_list = ['vanilla', 'chocolate', 'pecan', 'strawberry']  
for flavor in flavor_list:  
    print(f'{flavor} is delicious')
```

```
>>>
```

```
vanilla is delicious  
chocolate is delicious  
pecan is delicious  
strawberry is delicious
```

```
for i, flavor in enumerate(flavor_list):  
    print(f'{i + 1}: {flavor}')
```

```
>>>
```

```
1: vanilla
```

```
2: chocolate
```

```
3: pecan
```

```
4: strawberry
```

```
for i, flavor in enumerate(flavor_list, 1):  
    print(f'{i}: {flavor}')
```

```
names = ['Cecilia', 'Lise', 'Marie']  
counts = [len(n) for n in names]  
print(counts)
```

```
>>>
```

```
[7, 4, 5]
```

```
for name, count in zip(names, counts):  
    if count > max_count:  
        longest_name = name  
        max_count = count
```

```
names.append('Rosalind')  
for name, count in zip(names, counts):  
    print(name)
```

```
>>>
```

```
Cecilia
```

```
Lise
```

```
Marie
```

```
import itertools
```

```
for name, count in itertools.zip_longest(names, counts):  
    print(f'{name}: {count}')
```

```
>>>
```

```
Cecilia: 7
```

```
Lise: 4
```

```
Marie: 5
```

```
Rosalind: None
```



```
def coprime(a, b):  
    for i in range(2, min(a, b) + 1):  
        if a % i == 0 and b % i == 0:  
            return False  
    return True
```

```
assert coprime(4, 9)  
assert not coprime(3, 6)
```

```
def coprime_alternate(a, b):  
    is_coprime = True  
    for i in range(2, min(a, b) + 1):  
        if a % i == 0 and b % i == 0:  
            is_coprime = False  
            break  
    return is_coprime  
assert coprime_alternate(4, 9)  
assert not coprime_alternate(3, 6)
```

```
if count := fresh_fruit.get('lemon', 0):  
    make_lemonade(count)  
else:  
    out_of_stock()
```

```
if (count := fresh_fruit.get('apple', 0)) >= 4:  
    make_cider(count)  
else:  
    out_of_stock()
```

```
def slice_bananas(count):  
    ...  
  
class OutOfBananas(Exception):  
    pass  
  
def make_smoothies(count):  
    ...  
  
pieces = 0  
count = fresh_fruit.get('banana', 0)  
if count >= 2:  
    pieces = slice_bananas(count)  
  
try:  
    smoothies = make_smoothies(pieces)  
except OutOfBananas:  
    out_of_stock()
```

```
count = fresh_fruit.get('banana', 0)
if count >= 2:
    pieces = slice_bananas(count)
else:
    pieces = 0

try:
    smoothies = make_smoothies(pieces)
except OutOfBananas:
    out_of_stock()
```

```
pieces = 0
if (count := fresh_fruit.get('banana', 0)) >= 2:
    pieces = slice_bananas(count)

try:
    smoothies = make_smoothies(pieces)
except OutOfBananas:
    out_of_stock()
```

```
if (count := fresh_fruit.get('banana', 0)) >= 2:  
    pieces = slice_bananas(count)  
else:  
    pieces = 0  
  
try:  
    smoothies = make_smoothies(pieces)  
except OutOfBananas:  
    out_of_stock()
```



```
count = fresh_fruit.get('banana', 0)
if count >= 2:
    pieces = slice_bananas(count)
    to_enjoy = make_smoothies(pieces)
else:
    count = fresh_fruit.get('apple', 0)
    if count >= 4:
        to_enjoy = make_cider(count)
    else:
        count = fresh_fruit.get('lemon', 0)
        if count:
            to_enjoy = make_lemonade(count)
        else:
            to_enjoy = 'Nothing'
```

```
if (count := fresh_fruit.get('banana', 0)) >= 2:  
    pieces = slice_bananas(count)  
    to_enjoy = make_smoothies(pieces)  
elif (count := fresh_fruit.get('apple', 0)) >= 4:  
    to_enjoy = make_cider(count)  
elif count := fresh_fruit.get('lemon', 0):  
    to_enjoy = make_lemonade(count)  
else:  
    to_enjoy = 'Nothing'
```

```
def pick_fruit():  
    ...  
  
def make_juice(fruit, count):  
    ...  
  
bottles = []  
fresh_fruit = pick_fruit()  
while fresh_fruit:  
    for fruit, count in fresh_fruit.items():  
        batch = make_juice(fruit, count)  
        bottles.extend(batch)  
    fresh_fruit = pick_fruit()
```

```
bottles = []  
while True:                                # Loop  
    fresh_fruit = pick_fruit()  
    if not fresh_fruit:                    # And a half  
        break  
    for fruit, count in fresh_fruit.items():  
        batch = make_juice(fruit, count)  
        bottles.extend(batch)
```

```
bottles = []  
while fresh_fruit := pick_fruit():  
    for fruit, count in fresh_fruit.items():  
        batch = make_juice(fruit, count)  
        bottles.extend(batch)
```

```
a = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']  
print('Middle two: ', a[3:5])  
print('All but ends:', a[1:7])
```

```
>>>
```

```
Middle two:  ['d', 'e']
```

```
All but ends: ['b', 'c', 'd', 'e', 'f', 'g']
```

```
a[:]      # ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
a[:5]     # ['a', 'b', 'c', 'd', 'e']
a[:-1]    # ['a', 'b', 'c', 'd', 'e', 'f', 'g']
a[4:]     #           ['e', 'f', 'g', 'h']
a[-3:]    #           ['f', 'g', 'h']
a[2:5]    #           ['c', 'd', 'e']
a[2:-1]   #           ['c', 'd', 'e', 'f', 'g']
a[-3:-1]  #           ['f', 'g']
```

```
b = a[3:]  
print('Before: ', b)  
b[1] = 99  
print('After: ', b)  
print('No change:', a)
```

```
>>>  
Before:      ['d', 'e', 'f', 'g', 'h']  
After:       ['d', 99, 'f', 'g', 'h']  
No change:   ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
```



```
print('Before ', a)
a[2:7] = [99, 22, 14]
print('After ', a)
```

```
>>>
```

```
Before  ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
```

```
After   ['a', 'b', 99, 22, 14, 'h']
```

```
print('Before ', a)
```

```
a[2:3] = [47, 11]
```

```
print('After ', a)
```

```
>>>
```

```
Before  ['a', 'b', 99, 22, 14, 'h']
```

```
After   ['a', 'b', 47, 11, 22, 14, 'h']
```

```
b = a
print('Before a', a)
print('Before b', b)
a[:] = [101, 102, 103]
assert a is b          # Still the same list object
print('After a ', a)    # Now has different contents
print('After b ', b)    # Same list, so same contents as a
```

```
>>>
Before a ['a', 'b', 47, 11, 22, 14, 'h']
Before b ['a', 'b', 47, 11, 22, 14, 'h']
After a  [101, 102, 103]
After b  [101, 102, 103]
```

```
x = ['red', 'orange', 'yellow', 'green', 'blue', 'purple']
odds = x[::2]
evens = x[1::2]
print(odds)
print(evens)

>>>
['red', 'yellow', 'blue']
['orange', 'green', 'purple']
```

```
w = '寿司'
x = w.encode('utf-8')
y = x[::-1]
z = y.decode('utf-8')

>>>
Traceback ...
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xb8 in
position 0: invalid start byte
```

```
x = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']  
x[::2]    # ['a', 'c', 'e', 'g']  
x[::-2]   # ['h', 'f', 'd', 'b']
```

```
x[2::2]      # ['c', 'e', 'g']  
x[-2::-2]    # ['g', 'e', 'c', 'a']  
x[-2:2:-2]  # ['g', 'e']  
x[2:2:-2]   # []
```

```
y = x[::2]    # ['a', 'c', 'e', 'g']  
z = y[1:-1]   # ['c', 'e']
```



```
car_ages = [0, 9, 4, 8, 7, 20, 19, 1, 6, 15]
car_ages_descending = sorted(car_ages, reverse=True)
oldest, second_oldest = car_ages_descending
>>>
Traceback ...
ValueError: too many values to unpack (expected 2)
```

```
oldest = car_ages_descending[0]  
second_oldest = car_ages_descending[1]  
others = car_ages_descending[2:]  
print(oldest, second_oldest, others)
```

```
>>>
```

```
20 19 [15, 9, 8, 7, 6, 4, 1, 0]
```

```
oldest, second_oldest, *others = car_ages_descending  
print(oldest, second_oldest, others)
```

```
>>>
```

```
20 19 [15, 9, 8, 7, 6, 4, 1, 0]
```

```
oldest, *others, youngest = car_ages_descending  
print(oldest, youngest, others)
```

```
*others, second_youngest, youngest = car_ages_descending  
print(youngest, second_youngest, others)
```

```
>>>  
20 0 [19, 15, 9, 8, 7, 6, 4, 1]  
0 1 [20, 19, 15, 9, 8, 7, 6, 4]
```

```
*others = car_ages_descending
```

```
>>>
```

```
Traceback ...
```

```
SyntaxError: starred assignment target must be in a list or  
↳tuple
```

```
first, *middle, *second_middle, last = [1, 2, 3, 4]
```

```
>>>
```

```
Traceback ...
```

```
SyntaxError: two starred expressions in assignment
```

```
car_inventory = {
    'Downtown': ('Silver Shadow', 'Pinto', 'DMC'),
    'Airport': ('Skyline', 'Viper', 'Gremlin', 'Nova'),
}

((loc1, (best1, *rest1)),
 (loc2, (best2, *rest2))) = car_inventory.items()
print(f'Best at {loc1} is {best1}, {len(rest1)} others')
print(f'Best at {loc2} is {best2}, {len(rest2)} others')

>>>
Best at Downtown is Silver Shadow, 2 others
Best at Airport is Skyline, 3 others
```

```
def generate_csv():  
    yield ('Date', 'Make' , 'Model', 'Year', 'Price')  
    ...
```



```
all_csv_rows = list(generate_csv())
header = all_csv_rows[0]
rows = all_csv_rows[1:]
print('CSV Header:', header)
print('Row count: ', len(rows))

>>>
CSV Header: ('Date', 'Make', 'Model', 'Year', 'Price')
Row count: 200
```

```
it = generate_csv()
header, *rows = it
print('CSV Header:', header)
print('Row count: ', len(rows))

>>>
CSV Header: ('Date', 'Make', 'Model', 'Year', 'Price')
Row count: 200
```

```
class Tool:
    def __init__(self, name, weight):
        self.name = name
        self.weight = weight

    def __repr__(self):
        return f'Tool({self.name!r}, {self.weight})'

tools = [
    Tool('level', 3.5),
    Tool('hammer', 1.25),
    Tool('screwdriver', 0.5),
    Tool('chisel', 0.25),
]
```

```
tools.sort()
```

```
>>>
```

```
Traceback ...
```

```
TypeError: '<' not supported between instances of 'Tool' and  
'Tool'
```

```
print('Unsorted:', repr(tools))
tools.sort(key=lambda x: x.name)
print('\nSorted: ', tools)
```

```
>>>
```

```
Unsorted: [Tool('level',      3.5),
           Tool('hammer',    1.25),
           Tool('screwdriver', 0.5),
           Tool('chisel',    0.25)]
Sorted:   [Tool('chisel',    0.25),
           Tool('hammer',    1.25),
           Tool('level',     3.5),
           Tool('screwdriver', 0.5)]
```

```
tools.sort(key=lambda x: x.weight)
print('By weight:', tools)
```

```
>>>
```

```
By weight: [Tool('chisel',      0.25),
             Tool('screwdriver', 0.5),
             Tool('hammer',      1.25),
             Tool('level',       3.5)]
```

```
places = ['home', 'work', 'New York', 'Paris']
places.sort()
print('Case sensitive: ', places)
places.sort(key=lambda x: x.lower())
print('Case insensitive:', places)

>>>
Case sensitive:  ['New York', 'Paris', 'home', 'work']
Case insensitive: ['home', 'New York', 'Paris', 'work']
```

```
saw = (5, 'circular saw')  
jackhammer = (40, 'jackhammer')  
assert not (jackhammer < saw) # Matches expectations
```



```
drill = (4, 'drill')
sander = (4, 'sander')
assert drill[0] == sander[0] # Same weight
assert drill[1] < sander[1] # Alphabetically less
assert drill < sander # Thus, drill comes first
```

```
power_tools.sort(key=lambda x: (x.weight, x.name))  
print(power_tools)
```

```
>>>
```

```
[Tool('drill', 4),  
Tool('sander', 4),  
Tool('circular saw', 5),  
Tool('jackhammer', 40)]
```

```
power_tools.sort(key=lambda x: (x.weight, x.name),  
                  reverse=True) # Makes all criteria descending  
print(power_tools)  
  
>>>  
[Tool('jackhammer', 40),  
  Tool('circular saw', 5),  
  Tool('sander', 4),  
  Tool('drill', 4)]
```

```
power_tools.sort(key=lambda x: (-x.weight, x.name))  
print(power_tools)
```

```
>>>
```

```
[Tool('jackhammer', 40),  
Tool('circular saw', 5),  
Tool('drill', 4),  
Tool('sander', 4)]
```

```
power_tools.sort(key=lambda x: (x.weight, -x.name),  
                 reverse=True)
```

```
>>>
```

```
Traceback ...
```

```
TypeError: bad operand type for unary -: 'str'
```

```
power_tools.sort(key=lambda x: x.name)    # Name ascending
```

```
power_tools.sort(key=lambda x: x.weight, # Weight descending  
                  reverse=True)
```

```
print(power_tools)
```

```
>>>
```

```
[Tool('jackhammer', 40),  
 Tool('circular saw', 5),  
 Tool('drill', 4),  
 Tool('sander', 4)]
```

```
power_tools.sort(key=lambda x: x.name)  
print(power_tools)
```

```
>>>
```

```
[Tool('circular saw', 5),  
  Tool('drill', 4),  
  Tool('jackhammer', 40),  
  Tool('sander', 4)]
```

```
power_tools.sort(key=lambda x: x.weight,  
                  reverse=True)
```

```
print(power_tools)
```

```
>>>
```

```
[Tool('jackhammer', 40),  
 Tool('circular saw', 5),  
 Tool('drill', 4),  
 Tool('sander', 4)]
```



# Python 3.5

```
print(list(baby_names.keys()))
```

```
print(list(baby_names.values()))
```

```
print(list(baby_names.items()))
```

```
print(baby_names.popitem()) # Randomly chooses an item
```

```
>>>
```

```
['dog', 'cat']
```

```
['puppy', 'kitten']
```

```
[('dog', 'puppy'), ('cat', 'kitten')]
```

```
('dog', 'puppy')
```

```
print(list(baby_names.keys()))  
print(list(baby_names.values()))  
print(list(baby_names.items()))  
print(baby_names.popitem()) # Last item inserted
```

```
>>>
```

```
['cat', 'dog']  
['kitten', 'puppy']  
[('cat', 'kitten'), ('dog', 'puppy')]  
('dog', 'puppy')
```

```
# Python 3.5
```

```
def my_func(**kwargs):  
    for key, value in kwargs.items():  
        print('%s = %s' % (key, value))
```

```
my_func(goose='gosling', kangaroo='joey')
```

```
>>>
```

```
kangaroo = joey
```

```
goose = gosling
```

```
def my_func(**kwargs):  
    for key, value in kwargs.items():  
        print(f'{key} = {value}')  
  
my_func(goose='gosling', kangaroo='joey')  
  
>>>  
goose = gosling  
kangaroo = joey
```

```
# Python 3.5
```

```
class MyClass:
```

```
    def __init__(self):
```

```
        self.alligator = 'hatchling'
```

```
        self.elephant = 'calf'
```

```
a = MyClass()
```

```
for key, value in a.__dict__.items():
```

```
    print('%s = %s' % (key, value))
```

```
>>>
```

```
elephant = calf
```

```
alligator = hatchling
```

```
class MyClass:
    def __init__(self):
        self.alligator = 'hatchling'
        self.elephant = 'calf'

a = MyClass()
for key, value in a.__dict__.items():
    print(f'{key} = {value}')

>>>
alligator = hatchling
elephant = calf
```

```
def populate_ranks(votes, ranks):  
    names = list(votes.keys())  
    names.sort(key=votes.get, reverse=True)  
    for i, name in enumerate(names, 1):  
        ranks[name] = i
```

```
ranks = {}  
populate_ranks(votes, ranks)  
print(ranks)  
winner = get_winner(ranks)  
print(winner)  
  
>>>  
{'otter': 1, 'fox': 2, 'polar bear': 3}  
otter
```



```
from collections.abc import MutableMapping
```

```
class SortedDict(MutableMapping):
```

```
    def __init__(self):  
        self.data = {}
```

```
    def __getitem__(self, key):  
        return self.data[key]
```

```
    def __setitem__(self, key, value):  
        self.data[key] = value
```

```
    def __delitem__(self, key):  
        del self.data[key]
```

```
    def __iter__(self):  
        keys = list(self.data.keys())  
        keys.sort()  
        for key in keys:  
            yield key
```

```
    def __len__(self):  
        return len(self.data)
```

```
sorted_ranks = SortedDict()
populate_ranks(votes, sorted_ranks)
print(sorted_ranks.data)
winner = get_winner(sorted_ranks)
print(winner)

>>>
{'otter': 1, 'fox': 2, 'polar bear': 3}
fox
```

```
def get_winner(ranks):  
    for name, rank in ranks.items():  
        if rank == 1:  
            return name
```

```
winner = get_winner(sorted_ranks)  
print(winner)
```

```
>>>  
otter
```

```
def get_winner(ranks):  
    if not isinstance(ranks, dict):  
        raise TypeError('must provide a dict instance')  
    return next(iter(ranks))
```

```
get_winner(sorted_ranks)
```

```
>>>
```

```
Traceback ...
```

```
TypeError: must provide a dict instance
```

```

from typing import Dict, MutableMapping

def populate_ranks(votes: Dict[str, int],
                  ranks: Dict[str, int]) -> None:
    names = list(votes.keys())
    names.sort(key=votes.get, reverse=True)
    for i, name in enumerate(names, 1):
        ranks[name] = i

def get_winner(ranks: Dict[str, int]) -> str:
    return next(iter(ranks))

class SortedDict(MutableMapping[str, int]):
    ...

votes = {
    'otter': 1281,
    'polar bear': 587,
    'fox': 863,
}

sorted_ranks = SortedDict()
populate_ranks(votes, sorted_ranks)
print(sorted_ranks.data)
winner = get_winner(sorted_ranks)
print(winner)

```

```
$ python3 -m mypy --strict example.py
.../example.py:48: error: Argument 2 to "populate_ranks" has
➡ incompatible type "SortedDict"; expected "Dict[str, int]"
.../example.py:50: error: Argument 1 to "get_winner" has
➡ incompatible type "SortedDict"; expected "Dict[str, int]"
```

```
if (names := votes.get(key)) is None:  
    votes[key] = names = []
```

```
names.append(who)
```

```
visits = {  
    'Mexico': {'Tulum', 'Puerto Vallarta'},  
    'Japan': {'Hakone'},  
}
```



```
visits.setdefault('France', set()).add('Arles') # Short
```

```
if (japan := visits.get('Japan')) is None:      # Long  
    visits['Japan'] = japan = set()  
japan.add('Kyoto')
```

```
print(visits)
```

```
>>>
```

```
{'Mexico': {'Tulum', 'Puerto Vallarta'},  
 'Japan': {'Kyoto', 'Hakone'},  
 'France': {'Arles'}}
```

```
class Visits:
    def __init__(self):
        self.data = {}

    def add(self, country, city):
        city_set = self.data.setdefault(country, set())
        city_set.add(city)
```

```
visits = Visits()
visits.add('Russia', 'Yekaterinburg')
visits.add('Tanzania', 'Zanzibar')
print(visits.data)

>>>
{'Russia': {'Yekaterinburg'}, 'Tanzania': {'Zanzibar'}}
```

```
from collections import defaultdict

class Visits:
    def __init__(self):
        self.data = defaultdict(set)

    def add(self, country, city):
        self.data[country].add(city)

visits = Visits()
visits.add('England', 'Bath')
visits.add('England', 'London')
print(visits.data)

>>>
defaultdict(<class 'set'>, {'England': {'London', 'Bath'}})
```

```
pictures = {}  
path = 'profile_1234.png'  
  
if (handle := pictures.get(path)) is None:  
    try:  
        handle = open(path, 'a+b')  
    except OSError:  
        print(f'Failed to open path {path}')  
        raise  
    else:  
        pictures[path] = handle  
  
handle.seek(0)  
image_data = handle.read()
```

```
try:
    handle = pictures.setdefault(path, open(path, 'a+b'))
except OSError:
    print(f'Failed to open path {path}')
    raise
else:
    handle.seek(0)
    image_data = handle.read()
```

```
from collections import defaultdict

def open_picture(profile_path):
    try:
        return open(profile_path, 'a+b')
    except OSError:
        print(f'Failed to open path {profile_path}')
        raise

pictures = defaultdict(open_picture)
handle = pictures[path]
handle.seek(0)
image_data = handle.read()

>>>
Traceback ...
TypeError: open_picture() missing 1 required positional
argument: 'profile_path'
```

```
def get_stats(numbers):  
    minimum = min(numbers)  
    maximum = max(numbers)  
    return minimum, maximum
```

```
lengths = [63, 73, 72, 60, 67, 66, 71, 61, 72, 70]
```

```
minimum, maximum = get_stats(lengths) # Two return values
```

```
print(f'Min: {minimum}, Max: {maximum}')
```

```
>>>
```

```
Min: 60, Max: 73
```



```
def get_avg_ratio(numbers):  
    average = sum(numbers) / len(numbers)  
    scaled = [x / average for x in numbers]  
    scaled.sort(reverse=True)  
    return scaled  
  
longest, *middle, shortest = get_avg_ratio(lengths)  
  
print(f'Longest: {longest:>4.0%}')  
print(f'Shortest: {shortest:>4.0%}')  
  
>>>  
Longest: 108%  
Shortest: 89%
```

```
def get_stats(numbers):
    minimum = min(numbers)
    maximum = max(numbers)
    count = len(numbers)
    average = sum(numbers) / count

    sorted_numbers = sorted(numbers)
    middle = count // 2
    if count % 2 == 0:
        lower = sorted_numbers[middle - 1]
        upper = sorted_numbers[middle]
        median = (lower + upper) / 2
    else:
        median = sorted_numbers[middle]

    return minimum, maximum, average, median, count

minimum, maximum, average, median, count = get_stats(lengths)

print(f'Min: {minimum}, Max: {maximum}')
print(f'Average: {average}, Median: {median}, Count {count}')

>>>
Min: 60, Max: 73
Average: 67.5, Median: 68.5, Count 10
```

**# Correct:**

```
minimum, maximum, average, median, count = get_stats(lengths)
```

**# Oops! Median and average swapped:**

```
minimum, maximum, median, average, count = get_stats(lengths)
```

```
minimum, maximum, average, median, count = get_stats(  
    lengths)
```

```
minimum, maximum, average, median, count = \  
    get_stats(lengths)
```

```
(minimum, maximum, average,  
    median, count) = get_stats(lengths)
```

```
(minimum, maximum, average, median, count  
    ) = get_stats(lengths)
```

```
x, y = 0, 5
result = careful_divide(x, y)
if not result:
    print('Invalid inputs') # This runs! But shouldn't

>>>
Invalid inputs
```

```
success, result = careful_divide(x, y)
if not success:
    print('Invalid inputs')
```

```
def careful_divide(a, b):  
    try:  
        return a / b  
    except ZeroDivisionError as e:  
        raise ValueError('Invalid inputs')
```

```
x, y = 5, 2
try:
    result = careful_divide(x, y)
except ValueError:
    print('Invalid inputs')
else:
    print('Result is %.1f' % result)

>>>
Result is 2.5
```



```
def careful_divide(a: float, b: float) -> float:
    """Divides a by b.

    Raises:
        ValueError: When the inputs cannot be divided.
    """
    try:
        return a / b
    except ZeroDivisionError as e:
        raise ValueError('Invalid inputs')
```

```
def sort_priority2(numbers, group):  
    found = False  
    def helper(x):  
        if x in group:  
            found = True # Seems simple  
            return (0, x)  
        return (1, x)  
    numbers.sort(key=helper)  
    return found
```

```
found = sort_priority2(numbers, group)
print('Found:', found)
print(numbers)
```

```
>>>
```

```
Found: False
```

```
[2, 3, 5, 7, 1, 4, 6, 8]
```

```
foo = does_not_exist * 5
```

```
>>>
```

```
Traceback ...
```

```
NameError: name 'does_not_exist' is not defined
```

```
def sort_priority2(numbers, group):  
    found = False          # Scope: 'sort_priority2'  
    def helper(x):  
        if x in group:  
            found = True    # Scope: 'helper' -- Bad!  
            return (0, x)  
        return (1, x)  
    numbers.sort(key=helper)  
    return found
```

```
def log(message, values):  
    if not values:  
        print(message)  
    else:  
        values_str = ', '.join(str(x) for x in values)  
        print(f'{message}: {values_str}')
```

```
log('My numbers are', [1, 2])
```

```
log('Hi there', [])
```

```
>>>
```

```
My numbers are: 1, 2
```

```
Hi there
```

```
def log(message, *values): # The only difference
    if not values:
        print(message)
    else:
        values_str = ', '.join(str(x) for x in values)
        print(f'{message}: {values_str}')
```

```
log('My numbers are', 1, 2)
log('Hi there') # Much better
```

```
>>>
```

```
My numbers are: 1, 2
```

```
Hi there
```

```
def log(sequence, message, *values):
    if not values:
        print(f'{sequence} - {message}')
    else:
        values_str = ', '.join(str(x) for x in values)
        print(f'{sequence} - {message}: {values_str}')
```

```
log(1, 'Favorites', 7, 33)      # New with *args OK
log(1, 'Hi there')              # New message only OK
log('Favorite numbers', 7, 33)  # Old usage breaks
```

```
>>>
```

```
1 - Favorites: 7, 33
```

```
1 - Hi there
```

```
Favorite numbers - 7: 33
```



```
remainder(number=20, 7)
```

```
>>>
```

```
Traceback ...
```

```
SyntaxError: positional argument follows keyword argument
```

```
remainder(20, number=7)
```

```
>>>
```

```
Traceback ...
```

```
TypeError: remainder() got multiple values for argument
```

```
↳ 'number'
```

```
my_kwargs = {  
    'divisor': 7,  
}  
assert remainder(number=20, **my_kwargs) == 6
```

```
my_kwargs = {  
    'number': 20,  
}  
other_kwargs = {  
    'divisor': 7,  
}  
assert remainder(**my_kwargs, **other_kwargs) == 6
```

```
def print_parameters(**kwargs):  
    for key, value in kwargs.items():  
        print(f'{key} = {value}')
```

```
print_parameters(alpha=1.5, beta=9, gamma=4)
```

```
>>>  
alpha = 1.5  
beta = 9  
gamma = 4
```

```
def flow_rate(weight_diff, time_diff):  
    return weight_diff / time_diff
```

```
weight_diff = 0.5
```

```
time_diff = 3
```

```
flow = flow_rate(weight_diff, time_diff)
```

```
print(f'{flow:.3} kg per second')
```

```
>>>
```

```
0.167 kg per second
```

```
def flow_rate(weight_diff, time_diff, period):  
    return (weight_diff / time_diff) * period
```

```
flow_per_second = flow_rate(weight_diff, time_diff, 1)
```



```
def flow_rate(weight_diff, time_diff, period=1):  
    return (weight_diff / time_diff) * period
```

```
flow_per_second = flow_rate(weight_diff, time_diff)
flow_per_hour = flow_rate(weight_diff, time_diff, period=3600)
```

```
def flow_rate(weight_diff, time_diff,
               period=1, units_per_kg=1):
    return ((weight_diff * units_per_kg) / time_diff) * period
```



```
pounds_per_hour = flow_rate(weight_diff, time_diff, 3600, 2.2)
```

```
from time import sleep
from datetime import datetime

def log(message, when=datetime.now()):
    print(f'{when}: {message}')

log('Hi there!')
sleep(0.1)
log('Hello again!')

>>>
2019-07-06 14:06:15.120124: Hi there!
2019-07-06 14:06:15.120124: Hello again!
```

```
def log(message, when=None):
    """Log a message with a timestamp.

    Args:
        message: Message to print.
        when: datetime of when the message occurred.
            Defaults to the present time.
    """
    if when is None:
        when = datetime.now()
    print(f'{when}: {message}')
```

```
log('Hi there!')  
sleep(0.1)  
log('Hello again!')
```

```
>>>
```

```
2019-07-06 14:06:15.222419: Hi there!
```

```
2019-07-06 14:06:15.322555: Hello again!
```



```
def decode(data, default=None):
    """Load JSON data from a string.

    Args:
        data: JSON data to decode.
        default: Value to return if decoding fails.
                Defaults to an empty dictionary.
    """
    try:
        return json.loads(data)
    except ValueError:
        if default is None:
            default = {}
        return default
```

```
from typing import Optional
```

```
def log_typed(message: str,  
              when: Optional[datetime]=None) -> None:  
    """Log a message with a timestamp.
```

```
    Args:
```

```
        message: Message to print.
```

```
        when: datetime of when the message occurred.
```

```
        Defaults to the present time.
```

```
    """
```

```
    if when is None:
```

```
        when = datetime.now()
```

```
    print(f'{when}: {message}')
```

```
def safe_division(number, divisor,
                  ignore_overflow,
                  ignore_zero_division):
    try:
        return number / divisor
    except OverflowError:
        if ignore_overflow:
            return 0
        else:
            raise
    except ZeroDivisionError:
        if ignore_zero_division:
            return float('inf')
        else:
            raise
```

```
result = safe_division(1.0, 10**500, True, False)
print(result)
```

```
>>>
```

```
0
```

```
result = safe_division(1.0, 0, False, True)
print(result)
```

```
>>>
```

```
inf
```

```
def safe_division_b(number, divisor,  
                    ignore_overflow=False,      # Changed  
                    ignore_zero_division=False): # Changed  
    ...
```

```
result = safe_division_b(1.0, 10**500, ignore_overflow=True)
print(result)
```

```
result = safe_division_b(1.0, 0, ignore_zero_division=True)
print(result)
```

```
>>>
```

```
0
```

```
inf
```

```
assert safe_division_b(1.0, 10**500, True, False) == 0
```





```
safe_division_c(1.0, 10**500, True, False)
```

```
>>>
```

```
Traceback ...
```

```
TypeError: safe_division_c() takes 2 positional arguments but 4  
➡were given
```

```
result = safe_division_c(1.0, 0, ignore_zero_division=True)
assert result == float('inf')
try:
    result = safe_division_c(1.0, 0)
except ZeroDivisionError:
    pass # Expected
```

```
assert safe_division_c(number=2, divisor=5) == 0.4
assert safe_division_c(divisor=5, number=2) == 0.4
assert safe_division_c(2, divisor=5) == 0.4
```



```
safe_division_c(number=2, divisor=5)
```

```
>>>
```

```
Traceback ...
```

```
TypeError: safe_division_c() got an unexpected keyword argument
```

```
↳ 'number'
```



```
safe_division_d(numerator=2, denominator=5)
```

```
>>>
```

```
Traceback ...
```

```
TypeError: safe_division_d() got some positional-only arguments
```

```
↳ passed as keyword arguments: 'numerator, denominator'
```



```
def safe_division_e(numerator, denominator, /,
                    ndigits=10, *,
                    ignore_overflow=False,
                    ignore_zero_division=False):
    try:
        fraction = numerator / denominator
        return round(fraction, ndigits)
    except OverflowError:
        if ignore_overflow:
            return 0
        else:
            raise
    except ZeroDivisionError:
        if ignore_zero_division:
            return float('inf')
        else:
            raise
```

```
result = safe_division_e(22, 7)
print(result)
```

```
result = safe_division_e(22, 7, 5)
print(result)
```

```
result = safe_division_e(22, 7, ndigits=2)
print(result)
```

```
>>>
3.1428571429
3.14286
3.14
```

```
def trace(func):  
    def wrapper(*args, **kwargs):  
        result = func(*args, **kwargs)  
        print(f'{func.__name__}({args!r}, {kwargs!r}) '  
              f'-> {result!r}')  
        return result  
    return wrapper
```

```
@trace
def fibonacci(n):
    """Return the n-th Fibonacci number"""
    if n in (0, 1):
        return n
    return (fibonacci(n - 2) + fibonacci(n - 1))
```

```
print(fibonacci)
```

```
>>>
```

```
<function trace.<locals>.wrapper at 0x108955dc0>
```

```
help(fibonacci)
```

```
>>>
```

```
Help on function wrapper in module __main__:
```

```
wrapper(*args, **kwargs)
```

```
import pickle
```

```
pickle.dumps(fibonacci)
```

```
>>>
```

```
Traceback ...
```

```
AttributeError: Can't pickle local object 'trace.<locals>.  
↳wrapper'
```

```
help(fibonacci)
```

```
>>>
```

```
Help on function fibonacci in module __main__:
```

```
fibonacci(n)
```

```
    Return the n-th Fibonacci number
```



```
print(pickle.dumps(fibonacci))
```

```
>>>
```

```
b'\x80\x04\x95\x1a\x00\x00\x00\x00\x00\x00\x00\x8c\x08__main__\  
➡\x94\x8c\tfibonacci\x94\x93\x94.'
```

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
squares = []
for x in a:
    squares.append(x**2)
print(squares)

>>>
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
squares = [x**2 for x in a] # List comprehension  
print(squares)
```

```
>>>
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
even_squares = [x**2 for x in a if x % 2 == 0]  
print(even_squares)
```

```
>>>
```

```
[4, 16, 36, 64, 100]
```

```
alt = map(lambda x: x**2, filter(lambda x: x % 2 == 0, a))  
assert even_squares == list(alt)
```

```
even_squares_dict = {x: x**2 for x in a if x % 2 == 0}
threes_cubed_set = {x**3 for x in a if x % 3 == 0}
print(even_squares_dict)
print(threes_cubed_set)

>>>
{2: 4, 4: 16, 6: 36, 8: 64, 10: 100}
{216, 729, 27}
```

```
alt_dict = dict(map(lambda x: (x, x**2),  
                    filter(lambda x: x % 2 == 0, a)))  
alt_set = set(map(lambda x: x**3,  
                  filter(lambda x: x % 3 == 0, a)))
```

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]  
flat = [x for row in matrix for x in row]  
print(flat)
```

```
>>>
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```



```
squared = [[x**2 for x in row] for row in matrix]  
print(squared)
```

```
>>>
```

```
[[1, 4, 9], [16, 25, 36], [49, 64, 81]]
```

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
b = [x for x in a if x > 4 if x % 2 == 0]
c = [x for x in a if x > 4 and x % 2 == 0]
```

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
filtered = [[x for x in row if x % 3 == 0]
            for row in matrix if sum(row) >= 10]
print(filtered)

>>>
[[6], [9]]
```

```
stock = {
    'nails': 125,
    'screws': 35,
    'wingnuts': 8,
    'washers': 24,
}

order = ['screws', 'wingnuts', 'clips']

def get_batches(count, size):
    return count // size

result = {}
for name in order:
    count = stock.get(name, 0)
    batches = get_batches(count, 8)
    if batches:
        result[name] = batches

print(result)

>>>
{'screws': 4, 'wingnuts': 1}
```

```
found = {}
for name in order:
    found[name] = get_batches(stock.get(name, 0), 8)
print(found)

>>>
{'screws': 4, 'wingnuts': 1}
```

```
has_bug = {name: get_batches(stock.get(name, 0), 4)
            for name in order
            if get_batches(stock.get(name, 0), 8)}
```

```
print('Expected:', found)
print('Found:    ', has_bug)
```

```
>>>
```

```
Expected: {'screws': 4, 'wingnuts': 1}
```

```
Found:    {'screws': 8, 'wingnuts': 2}
```

```
found = {name: batches for name in order
         if (batches := get_batches(stock.get(name, 0), 8))}
```

```
result = {name: (tenth := count // 10)
          for name, count in stock.items() if tenth > 0}
```

```
>>>
```

```
Traceback ...
```

```
NameError: name 'tenth' is not defined
```



```
result = {name: tenth for name, count in stock.items()
          if (tenth := count // 10) > 0}
print(result)

>>>
{'nails': 12, 'screws': 3, 'washers': 2}
```

```
half = [(last := count // 2) for count in stock.values()]  
print(f'Last item of {half} is {last}')
```

```
>>>
```

```
Last item of [62, 17, 4, 12] is 12
```

```
for count in stock.values(): # Leaks loop variable
    pass
print(f'Last item of {list(stock.values())} is {count}')
>>>
Last item of [125, 35, 8, 24] is 24
```

```
half = [count // 2 for count in stock.values()]
print(half)    # Works
print(count)   # Exception because loop variable didn't leak

>>>
[62, 17, 4, 12]
Traceback ...
NameError: name 'count' is not defined
```

```
found = ((name, batches) for name in order
         if (batches := get_batches(stock.get(name, 0), 8)))
print(next(found))
print(next(found))

>>>
('screws', 4)
('wingnuts', 1)
```

```
def index_words(text):  
    result = []  
    if text:  
        result.append(0)  
    for index, letter in enumerate(text):  
        if letter == ' ':  
            result.append(index + 1)  
    return result
```

```
address = 'Four score and seven years ago...'
result = index_words(address)
print(result[:10])

>>>
[0, 5, 11, 15, 21, 27, 31, 35, 43, 51]
```

```
def index_words_iter(text):
    if text:
        yield 0
    for index, letter in enumerate(text):
        if letter == ' ':
            yield index + 1
```



```
result = list(index_words_iter(address))  
print(result[:10])
```

```
>>>
```

```
[0, 5, 11, 15, 21, 27, 31, 35, 43, 51]
```

```
with open('address.txt', 'r') as f:
    it = index_file(f)
    results = itertools.islice(it, 0, 10)
    print(list(results))
```

```
>>>
```

```
[0, 5, 11, 15, 21, 27, 31, 35, 43, 51]
```

```
visits = [15, 35, 80]
percentages = normalize(visits)
print(percentages)
assert sum(percentages) == 100.0

>>>
[11.538461538461538, 26.923076923076923, 61.53846153846154]
```

```
def normalize_copy(numbers):  
    numbers_copy = list(numbers) # Copy the iterator  
    total = sum(numbers_copy)  
    result = []  
    for value in numbers_copy:  
        percent = 100 * value / total  
        result.append(percent)  
    return result
```

```
it = read_visits('my_numbers.txt')
percentages = normalize_copy(it)
print(percentages)
assert sum(percentages) == 100.0

>>>
[11.538461538461538, 26.923076923076923, 61.53846153846154]
```

```
def normalize_func(get_iter):  
    total = sum(get_iter())    # New iterator  
    result = []  
    for value in get_iter():  # New iterator  
        percent = 100 * value / total  
        result.append(percent)  
    return result
```

```
path = 'my_numbers.txt'
percentages = normalize_func(lambda: read_visits(path))
print(percentages)
assert sum(percentages) == 100.0

>>>
[11.538461538461538, 26.923076923076923, 61.53846153846154]
```

```
visits = ReadVisits(path)
percentages = normalize(visits)
print(percentages)
assert sum(percentages) == 100.0

>>>
[11.538461538461538, 26.923076923076923, 61.53846153846154]
```



```
def normalize_defensive(numbers):  
    if iter(numbers) is numbers: # An iterator -- bad!  
        raise TypeError('Must supply a container')  
    total = sum(numbers)  
    result = []  
    for value in numbers:  
        percent = 100 * value / total  
        result.append(percent)  
    return result
```

```
from collections.abc import Iterator

def normalize_defensive(numbers):
    if isinstance(numbers, Iterator): # Another way to check
        raise TypeError('Must supply a container')
    total = sum(numbers)
    result = []
    for value in numbers:
        percent = 100 * value / total
        result.append(percent)
    return result
```

```
visits = [15, 35, 80]  
percentages = normalize_defensive(visits)  
assert sum(percentages) == 100.0
```

```
visits = ReadVisits(path)  
percentages = normalize_defensive(visits)  
assert sum(percentages) == 100.0
```

```
value = [len(x) for x in open('my_file.txt')]
print(value)
```

```
>>>
```

```
[100, 57, 15, 1, 12, 75, 5, 86, 89, 11]
```

```
it = (len(x) for x in open('my_file.txt'))  
print(it)
```

```
>>>
```

```
<generator object <genexpr> at 0x108993dd0>
```

```

import timeit

def child():
    for i in range(1_000_000):
        yield i

def slow():
    for i in child():
        yield i

def fast():
    yield from child()

baseline = timeit.timeit(
    stmt='for _ in slow(): pass',
    globals=globals(),
    number=50)
print(f'Manual nesting {baseline:.2f}s')

comparison = timeit.timeit(
    stmt='for _ in fast(): pass',
    globals=globals(),
    number=50)
print(f'Composed nesting {comparison:.2f}s')

reduction = -(comparison - baseline) / baseline
print(f'{reduction:.1%} less time')

>>>
Manual nesting 4.02s
Composed nesting 3.47s

```

13.5% less time

```
import math
```

```
def wave(amplitude, steps):  
    step_size = 2 * math.pi / steps  
    for step in range(steps):  
        radians = step * step_size  
        fraction = math.sin(radians)  
        output = amplitude * fraction  
        yield output
```



```
def transmit(output):  
    if output is None:  
        print(f'Output is None')  
    else:  
        print(f'Output: {output:>5.1f}')
```

```
def run(it):  
    for output in it:  
        transmit(output)
```

```
run(wave(3.0, 8))
```

```
>>>
```

```
Output: 0.0
```

```
Output: 2.1
```

```
Output: 3.0
```

```
Output: 2.1
```

```
Output: 0.0
```

```
Output: -2.1
```

```
Output: -3.0
```

```
Output: -2.1
```

```
def my_generator():
    received = yield 1
    print(f'received = {received}')

it = iter(my_generator())
output = next(it)          # Get first generator output
print(f'output = {output}')
try:
    next(it)                # Run generator until it exits
except StopIteration:
    pass

>>>
output = 1
received = None
```

```
it = iter(my_generator())
output = it.send(None) # Get first generator output
print(f'output = {output}')

try:
    it.send('hello!') # Send value into the generator
except StopIteration:
    pass

>>>
output = 1
received = hello!
```

```
def wave_modulating(steps):  
    step_size = 2 * math.pi / steps  
    amplitude = yield          # Receive initial amplitude  
    for step in range(steps):  
        radians = step * step_size  
        fraction = math.sin(radians)  
        output = amplitude * fraction  
        amplitude = yield output # Receive next amplitude
```

```
def run_modulating(it):
    amplitudes = [
        None, 7, 7, 7, 2, 2, 2, 2, 10, 10, 10, 10, 10]
    for amplitude in amplitudes:
        output = it.send(amplitude)
        transmit(output)
```

```
run_modulating(wave_modulating(12))
```

```
>>>
```

```
Output is None
```

```
Output: 0.0
```

```
Output: 3.5
```

```
Output: 6.1
```

```
Output: 2.0
```

```
Output: 1.7
```

```
Output: 1.0
```

```
Output: 0.0
```

```
Output: -5.0
```

```
Output: -8.7
```

```
Output: -10.0
```

```
Output: -8.7
```

```
Output: -5.0
```

```
def complex_wave_modulating():  
    yield from wave_modulating(3)  
    yield from wave_modulating(4)  
    yield from wave_modulating(5)
```

```
run_modulating(complex_wave_modulating())
```

```
>>>
```

```
Output is None
```

```
Output: 0.0
```

```
Output: 6.1
```

```
Output: -6.1
```

```
Output is None
```

```
Output: 0.0
```

```
Output: 2.0
```

```
Output: 0.0
```

```
Output: -10.0
```

```
Output is None
```

```
Output: 0.0
```

```
Output: 9.5
```

```
Output: 5.9
```

```
def wave_cascading(amplitude_it, steps):  
    step_size = 2 * math.pi / steps  
    for step in range(steps):  
        radians = step * step_size  
        fraction = math.sin(radians)  
        amplitude = next(amplitude_it) # Get next input  
        output = amplitude * fraction  
    yield output
```

```
def complex_wave_cascading(amplitude_it):  
    yield from wave_cascading(amplitude_it, 3)  
    yield from wave_cascading(amplitude_it, 4)  
    yield from wave_cascading(amplitude_it, 5)
```



```
def run_cascading():  
    amplitudes = [7, 7, 7, 2, 2, 2, 2, 10, 10, 10, 10, 10]  
    it = complex_wave_cascading(iter(amplitudes))  
    for amplitude in amplitudes:  
        output = next(it)  
        transmit(output)
```

```
run_cascading()
```

```
>>>
```

```
Output: 0.0
```

```
Output: 6.1
```

```
Output: -6.1
```

```
Output: 0.0
```

```
Output: 2.0
```

```
Output: 0.0
```

```
Output: -2.0
```

```
Output: 0.0
```

```
Output: 9.5
```

```
Output: 5.9
```

```
Output: -5.9
```

```
Output: -9.5
```

```
class MyError(Exception):
    pass

def my_generator():
    yield 1
    yield 2
    yield 3

it = my_generator()
print(next(it)) # Yield 1
print(next(it)) # Yield 2
print(it.throw(MyError('test error'))))

>>>
1
2
Traceback ...
MyError: test error
```

```
def my_generator():
    yield 1

    try:
        yield 2
    except MyError:
        print('Got MyError!')
    else:
        yield 3

    yield 4

it = my_generator()
print(next(it)) # Yield 1
print(next(it)) # Yield 2
print(it.throw(MyError('test error'))))

>>>
1
2
Got MyError!
4
```

```

def check_for_reset():
    # Poll for external event
    ...

def announce(remaining):
    print(f'{remaining} ticks remaining')

def run():
    it = timer(4)
    while True:
    try:
        if check_for_reset():
            current = it.throw(Reset())
        else:
            current = next(it)
    except StopIteration:
        break
    else:
        announce(current)

```

run()

>>>

```

3 ticks remaining
2 ticks remaining
1 ticks remaining
3 ticks remaining
2 ticks remaining
3 ticks remaining
2 ticks remaining
1 ticks remaining

```

0 ticks remaining

```
class Timer:
    def __init__(self, period):
        self.current = period
        self.period = period

    def reset(self):
        self.current = self.period

    def __iter__(self):
        while self.current:
            self.current -= 1
            yield self.current
```

```
it = itertools.chain([1, 2, 3], [4, 5, 6])  
print(list(it))
```

```
>>>
```

```
[1, 2, 3, 4, 5, 6]
```

```
it = itertools.cycle([1, 2])
result = [next(it) for _ in range (10)]
print(result)
>>>
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```



```
it1, it2, it3 = itertools.tee(['first', 'second'], 3)
print(list(it1))
print(list(it2))
print(list(it3))

>>>
['first', 'second']
['first', 'second']
['first', 'second']
```

```
keys = ['one', 'two', 'three']
values = [1, 2]

normal = list(zip(keys, values))
print('zip:          ', normal)

it = itertools.zip_longest(keys, values, fillvalue='nope')
longest = list(it)
print('zip_longest:', longest)

>>>
zip:          [('one', 1), ('two', 2)]
zip_longest: [('one', 1), ('two', 2), ('three', 'nope')]
```

```
values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
first_five = itertools.islice(values, 5)  
print('First five: ', list(first_five))
```

```
middle_odds = itertools.islice(values, 2, 8, 2)  
print('Middle odds:', list(middle_odds))
```

```
>>>
```

```
First five: [1, 2, 3, 4, 5]
```

```
Middle odds: [3, 5, 7]
```

```
values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
less_than_seven = lambda x: x < 7
it = itertools.takewhile(less_than_seven, values)
print(list(it))

>>>
[1, 2, 3, 4, 5, 6]
```

```
values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
less_than_seven = lambda x: x < 7
it = itertools.dropwhile(less_than_seven, values)
print(list(it))

>>>
[7, 8, 9, 10]
```

```
values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
evens = lambda x: x % 2 == 0
```

```
filter_result = filter(evens, values)
print('Filter:      ', list(filter_result))
```

```
filter_false_result = itertools.filterfalse(evens, values)
print('Filter false:', list(filter_false_result))
```

```
>>>
Filter:      [2, 4, 6, 8, 10]
Filter false: [1, 3, 5, 7, 9]
```

```
values = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sum_reduce = itertools.accumulate(values)
print('Sum: ', list(sum_reduce))
```

```
def sum_modulo_20(first, second):
    output = first + second
    return output % 20
```

```
modulo_reduce = itertools.accumulate(values, sum_modulo_20)
print('Modulo:', list(modulo_reduce))
```

```
>>>
```

```
Sum:      [1, 3, 6, 10, 15, 21, 28, 36, 45, 55]
```

```
Modulo: [1, 3, 6, 10, 15, 1, 8, 16, 5, 15]
```

```
single = itertools.product([1, 2], repeat=2)
print('Single: ', list(single))
```

```
multiple = itertools.product([1, 2], ['a', 'b'])
print('Multiple:', list(multiple))
```

```
>>>
```

```
Single:  [(1, 1), (1, 2), (2, 1), (2, 2)]
```

```
Multiple: [(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')]
```



```
it = itertools.permutations([1, 2, 3, 4], 2)
print(list(it))
```

```
>>>
```

```
[(1, 2),
 (1, 3),
 (1, 4),
 (2, 1),
 (2, 3),
 (2, 4),
 (3, 1),
 (3, 2),
 (3, 4),
 (4, 1),
 (4, 2),
 (4, 3)]
```

```
it = itertools.combinations([1, 2, 3, 4], 2)
print(list(it))
```

```
>>>
```

```
[(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]
```

```
it = itertools.combinations_with_replacement([1, 2, 3, 4], 2)
print(list(it))
```

```
>>>
```

```
[(1, 1),
 (1, 2),
 (1, 3),
 (1, 4),
 (2, 2),
 (2, 3),
 (2, 4),
 (3, 3),
 (3, 4),
 (4, 4)]
```

```
class SimpleGradebook:
    def __init__(self):
        self._grades = {}

    def add_student(self, name):
        self._grades[name] = []

    def report_grade(self, name, score):
        self._grades[name].append(score)

    def average_grade(self, name):
        grades = self._grades[name]
        return sum(grades) / len(grades)
```

```
book = SimpleGradebook()
book.add_student('Isaac Newton')
book.report_grade('Isaac Newton', 90)
book.report_grade('Isaac Newton', 95)
book.report_grade('Isaac Newton', 85)

print(book.average_grade('Isaac Newton'))

>>>
90.0
```

```
from collections import defaultdict

class BySubjectGradebook:
    def __init__(self):
        self._grades = {} # Outer dict

    def add_student(self, name):
        self._grades[name] = defaultdict(list) # Inner dict
```

```
def report_grade(self, name, subject, grade):  
    by_subject = self._grades[name]  
    grade_list = by_subject[subject]  
    grade_list.append(grade)  
  
def average_grade(self, name):  
    by_subject = self._grades[name]  
    total, count = 0, 0  
    for grades in by_subject.values():  
        total += sum(grades)  
        count += len(grades)  
    return total / count
```

```
book = BySubjectGradebook()
book.add_student('Albert Einstein')
book.report_grade('Albert Einstein', 'Math', 75)
book.report_grade('Albert Einstein', 'Math', 65)
book.report_grade('Albert Einstein', 'Gym', 90)
book.report_grade('Albert Einstein', 'Gym', 95)
print(book.average_grade('Albert Einstein'))

>>>
81.25
```



```
class WeightedGradebook:
    def __init__(self):
        self._grades = {}

    def add_student(self, name):
        self._grades[name] = defaultdict(list)

    def report_grade(self, name, subject, score, weight):
        by_subject = self._grades[name]
        grade_list = by_subject[subject]
        grade_list.append((score, weight))
```

```
def average_grade(self, name):  
    by_subject = self._grades[name]  
  
    score_sum, score_count = 0, 0  
    for subject, scores in by_subject.items():  
        subject_avg, total_weight = 0, 0  
        for score, weight in scores:  
            subject_avg += score * weight  
            total_weight += weight  
  
        score_sum += subject_avg / total_weight  
        score_count += 1  
  
    return score_sum / score_count
```

```
book = WeightedGradebook()
book.add_student('Albert Einstein')
book.report_grade('Albert Einstein', 'Math', 75, 0.05)
book.report_grade('Albert Einstein', 'Math', 65, 0.15)
book.report_grade('Albert Einstein', 'Math', 70, 0.80)
book.report_grade('Albert Einstein', 'Gym', 100, 0.40)
book.report_grade('Albert Einstein', 'Gym', 85, 0.60)
print(book.average_grade('Albert Einstein'))

>>>
80.25
```

```
grades = []  
grades.append((95, 0.45))  
grades.append((85, 0.55))  
total = sum(score * weight for score, weight in grades)  
total_weight = sum(weight for _, weight in grades)  
average_grade = total / total_weight
```

```
grades = []  
grades.append((95, 0.45, 'Great job'))  
grades.append((85, 0.55, 'Better next time'))  
total = sum(score * weight for score, weight, _ in grades)  
total_weight = sum(weight for _, weight, _ in grades)  
average_grade = total / total_weight
```

```
from collections import namedtuple
```

```
Grade = namedtuple('Grade', ('score', 'weight'))
```

```
class Subject:
    def __init__(self):
        self._grades = []

    def report_grade(self, score, weight):
        self._grades.append(Grade(score, weight))

    def average_grade(self):
        total, total_weight = 0, 0
        for grade in self._grades:
            total += grade.score * grade.weight
            total_weight += grade.weight
        return total / total_weight
```

```
class Student:
    def __init__(self):
        self._subjects = defaultdict(Subject)

    def get_subject(self, name):
        return self._subjects[name]

    def average_grade(self):
        total, count = 0, 0
        for subject in self._subjects.values():
            total += subject.average_grade()
            count += 1
        return total / count
```



```
class Gradebook:
    def __init__(self):
        self._students = defaultdict(Student)

    def get_student(self, name):
        return self._students[name]
```

```
book = Gradebook()
albert = book.get_student('Albert Einstein')
math = albert.get_subject('Math')
math.report_grade(75, 0.05)
math.report_grade(65, 0.15)
math.report_grade(70, 0.80)
gym = albert.get_subject('Gym')
gym.report_grade(100, 0.40)
gym.report_grade(85, 0.60)
print(albert.average_grade())

>>>
80.25
```

```
names = ['Socrates', 'Archimedes', 'Plato', 'Aristotle']
names.sort(key=len)
print(names)

>>>
['Plato', 'Socrates', 'Aristotle', 'Archimedes']
```

```
from collections import defaultdict
```

```
current = {'green': 12, 'blue': 3}
```

```
increments = [
```

```
    ('red', 5),
```

```
    ('blue', 17),
```

```
    ('orange', 9),
```

```
]
```

```
result = defaultdict(log_missing, current)
```

```
print('Before:', dict(result))
```

```
for key, amount in increments:
```

```
    result[key] += amount
```

```
print('After: ', dict(result))
```

```
>>>
```

```
Before: {'green': 12, 'blue': 3}
```

```
Key added
```

```
Key added
```

```
After:  {'green': 12, 'blue': 20, 'red': 5, 'orange': 9}
```

```
def increment_with_report(current, increments):  
    added_count = 0  
  
    def missing():  
        nonlocal added_count # Stateful closure  
        added_count += 1  
        return 0  
  
    result = defaultdict(missing, current)  
    for key, amount in increments:  
        result[key] += amount  
  
    return result, added_count
```

```
result, count = increment_with_report(current, increments)
assert count == 2
```

```
counter = CountMissing()
result = defaultdict(counter.missing, current) # Method ref
for key, amount in increments:
    result[key] += amount
assert counter.added == 2
```

```
counter = BetterCountMissing()
result = defaultdict(counter, current) # Relies on __call__
for key, amount in increments:
    result[key] += amount
assert counter.added == 2
```



```
class Worker:
    def __init__(self, input_data):
        self.input_data = input_data
        self.result = None

    def map(self):
        raise NotImplementedError

    def reduce(self, other):
        raise NotImplementedError
```

```
class LineCountWorker(Worker):  
    def map(self):  
        data = self.input_data.read()  
        self.result = data.count('\n')  
    def reduce(self, other):  
        self.result += other.result
```

```
import os
```

```
def generate_inputs(data_dir):  
    for name in os.listdir(data_dir):  
        yield PathInputData(os.path.join(data_dir, name))
```

```
def create_workers(input_list):  
    workers = []  
    for input_data in input_list:  
        workers.append(LineCountWorker(input_data))  
    return workers
```

```
from threading import Thread
```

```
def execute(workers):  
    threads = [Thread(target=w.map) for w in workers]  
    for thread in threads: thread.start()  
    for thread in threads: thread.join()  
  
    first, *rest = workers  
    for worker in rest:  
        first.reduce(worker)  
    return first.result
```

```
def mapreduce(data_dir):  
    inputs = generate_inputs(data_dir)  
    workers = create_workers(inputs)  
    return execute(workers)
```

```
import os
import random

def write_test_files(tmpdir):
    os.makedirs(tmpdir)
    for i in range(100):
        with open(os.path.join(tmpdir, str(i)), 'w') as f:
            f.write('\n' * random.randint(0, 100))

tmpdir = 'test_inputs'
write_test_files(tmpdir)

result = mapreduce(tmpdir)
print(f'There are {result} lines')

>>>
There are 4360 lines
```

```
class PathInputData(GenericInputData):
    ...

    @classmethod
    def generate_inputs(cls, config):
        data_dir = config['data_dir']
        for name in os.listdir(data_dir):
            yield cls(os.path.join(data_dir, name))
```



```
class GenericWorker:
    def __init__(self, input_data):
        self.input_data = input_data
        self.result = None

    def map(self):
        raise NotImplementedError

    def reduce(self, other):
        raise NotImplementedError

    @classmethod
    def create_workers(cls, input_class, config):
        workers = []
        for input_data in input_class.generate_inputs(config):
            workers.append(cls(input_data))
        return workers
```

```
class LineCountWorker(GenericWorker):  
    ...
```

```
def mapreduce(worker_class, input_class, config):  
    workers = worker_class.create_workers(input_class, config)  
    return execute(workers)
```

```
config = {'data_dir': tmpdir}
result = mapreduce(LineCountWorker, PathInputData, config)
print(f'There are {result} lines')
```

```
>>>
```

```
There are 4360 lines
```

```
class OneWay(MyBaseClass, TimesTwo, PlusFive):  
    def __init__(self, value):  
        MyBaseClass.__init__(self, value)  
        TimesTwo.__init__(self)  
        PlusFive.__init__(self)
```

```
foo = OneWay(5)
print('First ordering value is (5 * 2) + 5 =', foo.value)

>>>
First ordering value is (5 * 2) + 5 = 15
```

```
class AnotherWay(MyBaseClass, PlusFive, TimesTwo):  
    def __init__(self, value):  
        MyBaseClass.__init__(self, value)  
        TimesTwo.__init__(self)  
        PlusFive.__init__(self)
```

```
bar = AnotherWay(5)
print('Second ordering value is', bar.value)
```

```
>>>
```

```
Second ordering value is 15
```



```
class TimesSeven(MyBaseClass):  
    def __init__(self, value):  
        MyBaseClass.__init__(self, value)  
        self.value *= 7
```

```
class PlusNine(MyBaseClass):  
    def __init__(self, value):  
        MyBaseClass.__init__(self, value)  
        self.value += 9
```

```
class ThisWay(TimesSeven, PlusNine):
    def __init__(self, value):
        TimesSeven.__init__(self, value)
        PlusNine.__init__(self, value)

foo = ThisWay(5)
print('Should be (5 * 7) + 9 = 44 but is', foo.value)

>>>
Should be (5 * 7) + 9 = 44 but is 14
```

```
class GoodWay(TimesSevenCorrect, PlusNineCorrect):  
    def __init__(self, value):  
        super().__init__(value)
```

```
foo = GoodWay(5)
```

```
print('Should be 7 * (5 + 9) = 98 and is', foo.value)
```

```
>>>
```

```
Should be 7 * (5 + 9) = 98 and is 98
```

```
mro_str = '\n'.join(repr(cls) for cls in GoodWay.mro())  
print(mro_str)
```

```
>>>
```

```
<class '__main__.GoodWay'>  
<class '__main__.TimesSevenCorrect'>  
<class '__main__.PlusNineCorrect'>  
<class '__main__.MyBaseClass'>  
<class 'object'>
```

```
class ExplicitTrisect(MyBaseClass):  
    def __init__(self, value):  
        super(ExplicitTrisect, self).__init__(value)  
        self.value /= 3
```

```
class AutomaticTrisect(MyBaseClass):  
    def __init__(self, value):  
        super().__init__(value)  
        self.value /= 3
```

```
class ImplicitTrisect(MyBaseClass):  
    def __init__(self, value):  
        super().__init__(value)  
        self.value /= 3
```

```
assert ExplicitTrisect(9).value == 3  
assert AutomaticTrisect(9).value == 3  
assert ImplicitTrisect(9).value == 3
```

```
class ToDictMixin:
    def to_dict(self):
        return self._traverse_dict(self.__dict__)
```

```
def _traverse_dict(self, instance_dict):
    output = {}
    for key, value in instance_dict.items():
        output[key] = self._traverse(key, value)
    return output

def _traverse(self, key, value):
    if isinstance(value, ToDictMixin):
        return value.to_dict()
    elif isinstance(value, dict):
        return self._traverse_dict(value)
    elif isinstance(value, list):
        return [self._traverse(key, i) for i in value]
    elif hasattr(value, '__dict__'):
        return self._traverse_dict(value.__dict__)
    else:
        return value
```



```
class BinaryTree(ToDictMixin):  
    def __init__(self, value, left=None, right=None):  
        self.value = value  
        self.left = left  
        self.right = right
```

```
tree = BinaryTree(10,  
    left=BinaryTree(7, right=BinaryTree(9)),  
    right=BinaryTree(13, left=BinaryTree(11)))  
print(tree.to_dict())
```

```
>>>  
{'value': 10,  
  'left': {'value': 7,  
            'left': None,  
            'right': {'value': 9, 'left': None, 'right': None}},  
  'right': {'value': 13,  
            'left': {'value': 11, 'left': None, 'right': None},  
            'right': None}}
```

```
class BinaryTreeWithParent(BinaryTree):
    def __init__(self, value, left=None,
                  right=None, parent=None):
        super().__init__(value, left=left, right=right)
        self.parent = parent
```

```
def _traverse(self, key, value):  
    if (isinstance(value, BinaryTreeWithParent) and  
        key == 'parent'):  
        return value.value # Prevent cycles  
    else:  
        return super()._traverse(key, value)
```

```
root = BinaryTreeWithParent(10)
root.left = BinaryTreeWithParent(7, parent=root)
root.left.right = BinaryTreeWithParent(9, parent=root.left)
print(root.to_dict())
```

```
>>>
{'value': 10,
 'left': {'value': 7,
          'left': None,
          'right': {'value': 9,
                    'left': None,
                    'right': None,
                    'parent': 7},
          'parent': 10},
 'right': None,
 'parent': None}
```

```
class NamedSubTree(ToDictMixin):
    def __init__(self, name, tree_with_parent):
        self.name = name
        self.tree_with_parent = tree_with_parent

my_tree = NamedSubTree('foobar', root.left.right)
print(my_tree.to_dict()) # No infinite loop

>>>
{'name': 'foobar',
 'tree_with_parent': {'value': 9,
                       'left': None,
                       'right': None,
                       'parent': 7}}
```

```
import json
```

```
class JsonMixin:
```

```
    @classmethod
```

```
    def from_json(cls, data):
```

```
        kwargs = json.loads(data)
```

```
        return cls(**kwargs)
```

```
    def to_json(self):
```

```
        return json.dumps(self.to_dict())
```

```
class DatacenterRack(ToDictMixin, JsonMixin):
    def __init__(self, switch=None, machines=None):
        self.switch = Switch(**switch)
        self.machines = [
            Machine(**kwargs) for kwargs in machines]

class Switch(ToDictMixin, JsonMixin):
    def __init__(self, ports=None, speed=None):
        self.ports = ports
        self.speed = speed

class Machine(ToDictMixin, JsonMixin):
    def __init__(self, cores=None, ram=None, disk=None):
        self.cores = cores
        self.ram = ram
        self.disk = disk
```



```
serialized = """{
    "switch": {"ports": 5, "speed": 1e9},
    "machines": [
        {"cores": 8, "ram": 32e9, "disk": 5e12},
        {"cores": 4, "ram": 16e9, "disk": 1e12},
        {"cores": 2, "ram": 4e9, "disk": 500e9}
    ]
}"""
```

```
deserialized = DatacenterRack.from_json(serialized)
roundtrip = deserialized.to_json()
assert json.loads(serialized) == json.loads(roundtrip)
```

```
assert foo.get_private_field() == 10
```

```
foo.__private_field
```

```
>>>
```

```
Traceback ...
```

```
AttributeError: 'MyObject' object has no attribute
```

```
↳ '__private_field'
```

```
class MyOtherObject:
    def __init__(self):
        self.__private_field = 71

    @classmethod
    def get_private_field_of_instance(cls, instance):
        return instance.__private_field

bar = MyOtherObject()
assert MyOtherObject.get_private_field_of_instance(bar) == 71
```

```
class MyParentObject:
    def __init__(self):
        self.__private_field = 71
```

```
class MyChildObject(MyParentObject):
    def get_private_field(self):
        return self.__private_field
```

```
baz = MyChildObject()
baz.get_private_field()
```

```
>>>
```

```
Traceback ...
```

```
AttributeError: 'MyChildObject' object has no attribute
```

```
↳ '_MyChildObject__private_field'
```

```
assert baz._MyParentObject__private_field == 71
```

```
print(baz.__dict__)
```

```
>>>
```

```
{'_MyParentObject__private_field': 71}
```

```
class MyIntegerSubclass(MyStringClass):  
    def get_value(self):  
        return int(self._MyStringClass__value)  
foo = MyIntegerSubclass('5')  
assert foo.get_value() == 5
```



```
class MyBaseClass:
    def __init__(self, value):
        self.__value = value

    def get_value(self):
        return self.__value

class MyStringClass(MyBaseClass):
    def get_value(self):
        return str(super().get_value()) # Updated

class MyIntegerSubclass(MyStringClass):
    def get_value(self):
        return int(self._MyStringClass__value) # Not updated
```

```
foo = MyIntegerSubclass(5)
foo.get_value()
```

```
>>>
```

```
Traceback ...
```

```
AttributeError: 'MyIntegerSubclass' object has no attribute
↳ '_MyStringClass__value'
```

```
class MyStringClass:
    def __init__(self, value):
        # This stores the user-supplied value for the object.
        # It should be coercible to a string. Once assigned in
        # the object it should be treated as immutable.

        self._value = value

    ...
```

```
class ApiClass:
    def __init__(self):
        self._value = 5

    def get(self):
        return self._value

class Child(ApiClass):
    def __init__(self):
        super().__init__()
        self._value = 'hello' # Conflicts

a = Child()
print(f'{a.get()} and {a._value} should be different')

>>>
hello and hello should be different
```

```
class ApiClass:
    def __init__(self):
        self.__value = 5          # Double underscore

    def get(self):
        return self.__value      # Double underscore

class Child(ApiClass):
    def __init__(self):
        super().__init__()
        self._value = 'hello'    # OK!

    a = Child()
    print(f'{a.get()} and {a._value} are different')

>>>
5 and hello are different
```

```
class FrequencyList(list):  
    def __init__(self, members):  
        super().__init__(members)  
  
    def frequency(self):  
        counts = {}  
        for item in self:  
            counts[item] = counts.get(item, 0) + 1  
        return counts
```

```
foo = FrequencyList(['a', 'b', 'a', 'c', 'b', 'a', 'd'])
print('Length is', len(foo))
foo.pop()
print('After pop:', repr(foo))
print('Frequency:', foo.frequency())

>>>
Length is 7
After pop: ['a', 'b', 'a', 'c', 'b', 'a']
Frequency: {'a': 3, 'b': 2, 'c': 1}
```

```
class BinaryNode:
    def __init__(self, value, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right
```



```
class IndexableNode(BinaryNode):
    def _traverse(self):
        if self.left is not None:
            yield from self.left._traverse()
        yield self
        if self.right is not None:
            yield from self.right._traverse()

    def __getitem__(self, index):
        for i, item in enumerate(self._traverse()):
            if i == index:
                return item.value
        raise IndexError(f'Index {index} is out of range')
```

```
tree = IndexableNode(  
    10,  
    left=IndexableNode(  
        5,  
        left=IndexableNode(2),  
        right=IndexableNode(  
            6,  
            right=IndexableNode(7))),  
    right=IndexableNode(  
        15,  
        left=IndexableNode(11)))
```

```
print('LRR is', tree.left.right.right.value)
print('Index 0 is', tree[0])
print('Index 1 is', tree[1])
print('11 in the tree?', 11 in tree)
print('17 in the tree?', 17 in tree)
print('Tree is', list(tree))
```

```
>>>
```

```
LRR is 7
```

```
Index 0 is 2
```

```
Index 1 is 5
```

```
11 in the tree? True
```

```
17 in the tree? False
```

```
Tree is [2, 5, 6, 7, 10, 11, 15]
```

```
len(tree)
```

```
>>>
```

```
Traceback ...
```

```
TypeError: object of type 'IndexableNode' has no len()
```

```
class SequenceNode(IndexableNode):
    def __len__(self):
        for count, _ in enumerate(self._traverse(), 1):
            pass
        return count

    tree = SequenceNode(
        10,
        left=SequenceNode(
            5,
            left=SequenceNode(2),
            right=SequenceNode(
                6,
                right=SequenceNode(7))),
        right=SequenceNode(
            15,
            left=SequenceNode(11))
    )

print('Tree length is', len(tree))

>>>
Tree length is 7
```

```
from collections.abc import Sequence
```

```
class BadType(Sequence):  
    pass
```

```
foo = BadType()
```

```
>>>
```

```
Traceback ...
```

```
TypeError: Can't instantiate abstract class BadType with  
↳ abstract methods __getitem__, __len__
```

```
class BetterNode(SequenceNode, Sequence):
    pass

tree = BetterNode(
    10,
    left=BetterNode(
        5,
        left=BetterNode(2),
        right=BetterNode(
            6,
            right=BetterNode(7))),
    right=BetterNode(
        15,
        left=BetterNode(11))
)

print('Index of 7 is', tree.index(7))
print('Count of 10 is', tree.count(10))

>>>
Index of 7 is 3
Count of 10 is 1
```

```
class VoltageResistance(Resistor):
    def __init__(self, ohms):
        super().__init__(ohms)
        self._voltage = 0

    @property
    def voltage(self):
        return self._voltage

    @voltage.setter
    def voltage(self, voltage):
        self._voltage = voltage
        self.current = self._voltage / self.ohms
```



```
r2 = VoltageResistance(1e3)
print(f'Before: {r2.current:.2f} amps')
r2.voltage = 10
print(f'After: {r2.current:.2f} amps')
```

```
>>>
```

```
Before: 0.00 amps
```

```
After: 0.01 amps
```

```
class BoundedResistance(Resistor):
    def __init__(self, ohms):
        super().__init__(ohms)

    @property
    def ohms(self):
        return self._ohms

    @ohms.setter
    def ohms(self, ohms):
        if ohms <= 0:
            raise ValueError(f'ohms must be > 0; got {ohms}')
        self._ohms = ohms
```

```
BoundedResistance(-5)
```

```
>>>
```

```
Traceback ...
```

```
ValueError: ohms must be > 0; got -5
```

```
class FixedResistance(Resistor):
    def __init__(self, ohms):
        super().__init__(ohms)

    @property
    def ohms(self):
        return self._ohms

    @ohms.setter
    def ohms(self, ohms):
        if hasattr(self, '_ohms'):
            raise AttributeError("Ohms is immutable")
        self._ohms = ohms
```

```
class MysteriousResistor(Resistor):  
    @property  
    def ohms(self):  
        self.voltage = self._ohms * self.current  
        return self._ohms  
  
    @ohms.setter  
    def ohms(self, ohms):  
        self._ohms = ohms
```

```
from datetime import datetime, timedelta

class Bucket:
    def __init__(self, period):
        self.period_delta = timedelta(seconds=period)
        self.reset_time = datetime.now()
        self.quota = 0

    def __repr__(self):
        return f'Bucket(quota={self.quota})'
```

```
def fill(bucket, amount):  
    now = datetime.now()  
    if (now - bucket.reset_time) > bucket.period_delta:  
        bucket.quota = 0  
        bucket.reset_time = now  
    bucket.quota += amount
```

```
def deduct(bucket, amount):  
    now = datetime.now()  
    if (now - bucket.reset_time) > bucket.period_delta:  
        return False # Bucket hasn't been filled this period  
    if bucket.quota - amount < 0:  
        return False # Bucket was filled, but not enough  
    bucket.quota -= amount  
    return True      # Bucket had enough, quota consumed
```



```
if deduct(bucket, 99):  
    print('Had 99 quota')  
else:  
    print('Not enough for 99 quota')  
print(bucket)
```

```
>>>
```

```
Had 99 quota
```

```
Bucket(quota=1)
```

```
class NewBucket:
    def __init__(self, period):
        self.period_delta = timedelta(seconds=period)
        self.reset_time = datetime.now()
        self.max_quota = 0
        self.quota_consumed = 0

    def __repr__(self):
        return (f'NewBucket(max_quota={self.max_quota}, '
                f'quota_consumed={self.quota_consumed})')
```

```
@property  
def quota(self):  
    return self.max_quota - self.quota_consumed
```

```
@quota.setter
def quota(self, amount):
    delta = self.max_quota - amount
    if amount == 0:
        # Quota being reset for a new period
        self.quota_consumed = 0
        self.max_quota = 0
    elif delta < 0:
        # Quota being filled for the new period
        assert self.quota_consumed == 0
        self.max_quota = amount
    else:
        # Quota being consumed during the period
        assert self.max_quota >= self.quota_consumed
        self.quota_consumed += delta
```

```
bucket = NewBucket(60)
print('Initial', bucket)
fill(bucket, 100)
print('Filled', bucket)

if deduct(bucket, 99):
    print('Had 99 quota')
else:
    print('Not enough for 99 quota')
print('Now', bucket)

if deduct(bucket, 3):
    print('Had 3 quota')
else:
    print('Not enough for 3 quota')

print('Still', bucket)

>>>
Initial NewBucket(max_quota=0, quota_consumed=0)
Filled NewBucket(max_quota=100, quota_consumed=0)
Had 99 quota
Now NewBucket(max_quota=100, quota_consumed=99)
Not enough for 3 quota
Still NewBucket(max_quota=100, quota_consumed=99)
```

```
class Homework:
    def __init__(self):
        self._grade = 0

    @property
    def grade(self):
        return self._grade

    @grade.setter
    def grade(self, value):
        if not (0 <= value <= 100):
            raise ValueError(
                'Grade must be between 0 and 100')
        self._grade = value
```

```
class Exam:
    def __init__(self):
        self._writing_grade = 0
        self._math_grade = 0

    @staticmethod
    def _check_grade(value):
        if not (0 <= value <= 100):
            raise ValueError(
                'Grade must be between 0 and 100')
```

```
class Grade:
    def __get__(self, instance, instance_type):
        ...

    def __set__(self, instance, value):
        ...

class Exam:
    # Class attributes
    math_grade = Grade()
    writing_grade = Grade()
    science_grade = Grade()
```



```
Exam.__dict__['writing_grade'].__set__(exam, 40)
```

```
Exam.__dict__['writing_grade'].__get__(exam, Exam)
```

```
class Grade:
    def __init__(self):
        self._value = 0

    def __get__(self, instance, instance_type):
        return self._value

    def __set__(self, instance, value):
        if not (0 <= value <= 100):
            raise ValueError(
                'Grade must be between 0 and 100')
        self._value = value
```

```
class Exam:
    math_grade = Grade()
    writing_grade = Grade()
    science_grade = Grade()

first_exam = Exam()
first_exam.writing_grade = 82
first_exam.science_grade = 99
print('Writing', first_exam.writing_grade)
print('Science', first_exam.science_grade)

>>>
Writing 82
Science 99
```

```
second_exam = Exam()  
second_exam.writing_grade = 75  
print(f'Second {second_exam.writing_grade} is right')  
print(f'First {first_exam.writing_grade} is wrong; '  
      f'should be 82')
```

```
>>>  
Second 75 is right  
First 75 is wrong; should be 82
```

```
class Grade:
    def __init__(self):
        self._values = {}

    def __get__(self, instance, instance_type):
        if instance is None:
            return self
        return self._values.get(instance, 0)
    def __set__(self, instance, value):
        if not (0 <= value <= 100):
            raise ValueError(
                'Grade must be between 0 and 100')
        self._values[instance] = value
```

```
from weakref import WeakKeyDictionary

class Grade:
    def __init__(self):
        self._values = WeakKeyDictionary()

    def __get__(self, instance, instance_type):
        ...

    def __set__(self, instance, value):
        ...
```

```
class Exam:
    math_grade = Grade()
    writing_grade = Grade()
    science_grade = Grade()

first_exam = Exam()
first_exam.writing_grade = 82
second_exam = Exam()
second_exam.writing_grade = 75
print(f'First {first_exam.writing_grade} is right')
print(f'Second {second_exam.writing_grade} is right')
>>>
First 82 is right
Second 75 is right
```



```
data = LazyRecord()
print('Before:', data.__dict__)
print('foo:    ', data.foo)
print('After:  ', data.__dict__)

>>>
Before: {'exists': 5}
foo:    Value for foo
After:  {'exists': 5, 'foo': 'Value for foo'}
```

```
class LoggingLazyRecord(LazyRecord):
    def __getattr__(self, name):
        print(f'* Called __getattr__({name!r}), '
              f'populating instance dictionary')
        result = super().__getattr__(name)
        print(f'* Returning {result!r}')
        return result
```

```
data = LoggingLazyRecord()
print('exists:      ', data.exists)
print('First foo:   ', data.foo)
print('Second foo: ', data.foo)
```

```
>>>
exists:      5
* Called __getattr__('foo'), populating instance dictionary
* Returning 'Value for foo'
First foo:   Value for foo
Second foo:  Value for foo
```

```

class ValidatingRecord:
    def __init__(self):
        self.exists = 5

    def __getattr__(self, name):
        print(f'* Called __getattr__({name!r})')
        try:
            value = super().__getattr__(name)
            print(f'* Found {name!r}, returning {value!r}')
            return value
        except AttributeError:
            value = f'Value for {name}'
            print(f'* Setting {name!r} to {value!r}')
            setattr(self, name, value)
            return value

```

```

data = ValidatingRecord()
print('exists: ', data.exists)
print('First foo: ', data.foo)
print('Second foo: ', data.foo)

```

```

>>>
* Called __getattr__('exists')
* Found 'exists', returning 5
exists:      5
* Called __getattr__('foo')
* Setting 'foo' to 'Value for foo'
First foo:   Value for foo
* Called __getattr__('foo')
* Found 'foo', returning 'Value for foo'
Second foo:  Value for foo

```

```
class MissingPropertyRecord:
    def __getattr__(self, name):
        if name == 'bad_name':
            raise AttributeError(f'{name} is missing')
        ...
```

```
data = MissingPropertyRecord()
data.bad_name
```

```
>>>
```

```
Traceback ...
```

```
AttributeError: bad_name is missing
```

```
data = LoggingLazyRecord() # Implements __getattr__
print('Before:          ', data.__dict__)
print('Has first foo:   ', hasattr(data, 'foo'))
print('After:          ', data.__dict__)
print('Has second foo: ', hasattr(data, 'foo'))

>>>
Before:          {'exists': 5}
* Called __getattr__('foo'), populating instance dictionary
* Returning 'Value for foo'
Has first foo:   True
After:          {'exists': 5, 'foo': 'Value for foo'}
Has second foo: True
```

```
data = ValidatingRecord() # Implements __getattribute__
print('Has first foo: ', hasattr(data, 'foo'))
print('Has second foo: ', hasattr(data, 'foo'))
```

```
>>>
```

```
* Called __getattribute__('foo')
```

```
* Setting 'foo' to 'Value for foo'
```

```
Has first foo: True
```

```
* Called __getattribute__('foo')
```

```
* Found 'foo', returning 'Value for foo'
```

```
Has second foo: True
```

```
class SavingRecord:
    def __setattr__(self, name, value):
        # Save some data for the record
        ...
        super().__setattr__(name, value)
```

```
class LoggingSavingRecord(SavingRecord):
    def __setattr__(self, name, value):
        print(f'* Called __setattr__({name!r}, {value!r})')
        super().__setattr__(name, value)
```

```
data = LoggingSavingRecord()
print('Before: ', data.__dict__)
data.foo = 5
print('After: ', data.__dict__)
data.foo = 7
print('Finally:', data.__dict__)
```

```
>>>
```

```
Before: {}
* Called __setattr__('foo', 5)
After: {'foo': 5}
* Called __setattr__('foo', 7)
Finally: {'foo': 7}
```



```
class BrokenDictionaryRecord:
    def __init__(self, data):
        self._data = {}
    def __getattr__(self, name):
        print(f'* Called __getattr__({name!r})')
        return self._data[name]
```

```
data = BrokenDictionaryRecord({'foo': 3})  
data.foo
```

```
>>>
```

```
* Called __getattr__('foo')
```

```
* Called __getattr__('_data')
```

```
* Called __getattr__('_data')
```

```
* Called __getattr__('_data')
```

```
...
```

```
Traceback ...
```

```
RecursionError: maximum recursion depth exceeded while calling
```

```
↳ a Python object
```

```
class DictionaryRecord:
    def __init__(self, data):
        self._data = data

    def __getattr__(self, name):
        print(f'* Called __getattr__({name!r})')
        data_dict = super().__getattr__('_data')
        return data_dict[name]

data = DictionaryRecord({'foo': 3})
print('foo: ', data.foo)

>>>
* Called __getattr__('foo')
foo: 3
```

```
class Meta(type):
    def __new__(meta, name, bases, class_dict):
        print(f'* Running {meta}.__new__ for {name}')
        print('Bases:', bases)
        print(class_dict)
        return type.__new__(meta, name, bases, class_dict)

class MyClass(metaclass=Meta):
    stuff = 123

    def foo(self):
        pass

class MySubclass(MyClass):
    other = 567

    def bar(self):
        pass
```

```
>>>
* Running <class '__main__.Meta'>.__new__ for MyClass
Bases: ()
{'__module__': '__main__',
  '__qualname__': 'MyClass',
  'stuff': 123,
  'foo': <function MyClass.foo at 0x105a05280>}
* Running <class '__main__.Meta'>.__new__ for MySubclass
Bases: (<class '__main__.MyClass'>,)
{'__module__': '__main__',
  '__qualname__': 'MySubclass',
  'other': 567,
  'bar': <function MySubclass.bar at 0x105a05310>}
```

```
class ValidatePolygon(type):
    def __new__(meta, name, bases, class_dict):
        # Only validate subclasses of the Polygon class
        if bases:
            if class_dict['sides'] < 3:
                raise ValueError('Polygons need 3+ sides')
        return type.__new__(meta, name, bases, class_dict)

class Polygon(metaclass=ValidatePolygon):
    sides = None # Must be specified by subclasses

    @classmethod
    def interior_angles(cls):
        return (cls.sides - 2) * 180

class Triangle(Polygon):
    sides = 3

class Rectangle(Polygon):
    sides = 4

class Nonagon(Polygon):
    sides = 9

assert Triangle.interior_angles() == 180
assert Rectangle.interior_angles() == 360
assert Nonagon.interior_angles() == 1260
```

```
class BetterPolygon:
    sides = None # Must be specified by subclasses

    def __init_subclass__(cls):
        super().__init_subclass__()
        if cls.sides < 3:
            raise ValueError('Polygons need 3+ sides')
    @classmethod
    def interior_angles(cls):
        return (cls.sides - 2) * 180

class Hexagon(BetterPolygon):
    sides = 6

assert Hexagon.interior_angles() == 720
```

```
class ValidateFilled(type):
    def __new__(meta, name, bases, class_dict):
        # Only validate subclasses of the Filled class
        if bases:
            if class_dict['color'] not in ('red', 'green'):
                raise ValueError('Fill color must be supported')
        return type.__new__(meta, name, bases, class_dict)

class Filled(metaclass=ValidateFilled):
    color = None # Must be specified by subclasses
```



```
class RedPentagon(Filled, Polygon):  
    color = 'red'  
    sides = 5
```

```
>>>
```

```
Traceback ...
```

```
TypeError: metaclass conflict: the metaclass of a derived  
↳class must be a (non-strict) subclass of the metaclasses  
↳of all its bases
```

```

class ValidatePolygon(type):
    def __new__(meta, name, bases, class_dict):
        # Only validate non-root classes
        if not class_dict.get('is_root'):
            if class_dict['sides'] < 3:
                raise ValueError('Polygons need 3+ sides')
        return type.__new__(meta, name, bases, class_dict)

class Polygon(metaclass=ValidatePolygon):
    is_root = True
    sides = None # Must be specified by subclasses

class ValidateFilledPolygon(ValidatePolygon):
    def __new__(meta, name, bases, class_dict):
        # Only validate non-root classes
        if not class_dict.get('is_root'):
            if class_dict['color'] not in ('red', 'green'):
                raise ValueError('Fill color must be supported')
        return super().__new__(meta, name, bases, class_dict)

class FilledPolygon(Polygon, metaclass=ValidateFilledPolygon):
    is_root = True
    color = None # Must be specified by subclasses

```

```
class OrangePentagon(FilledPolygon):  
    color = 'orange'  
    sides = 5
```

```
>>>
```

```
Traceback ...
```

```
ValueError: Fill color must be supported
```

```
class Filled:
    color = None # Must be specified by subclasses

    def __init_subclass__(cls):
        super().__init_subclass__()
        if cls.color not in ('red', 'green', 'blue'):
            raise ValueError('Fills need a valid color')
```

```
import json
```

```
class Serializable:
```

```
    def __init__(self, *args):  
        self.args = args
```

```
    def serialize(self):  
        return json.dumps({'args': self.args})
```

```
class Point2D(Serializable):
    def __init__(self, x, y):
        super().__init__(x, y)
        self.x = x
        self.y = y

    def __repr__(self):
        return f'Point2D({self.x}, {self.y})'
```

```
point = Point2D(5, 3)
print('Object:      ', point)
print('Serialized:', point.serialize())
```

```
>>>
```

```
Object:      Point2D(5, 3)
Serialized: {"args": [5, 3]}
```

```
class Deserializable(Serializable):  
    @classmethod  
    def deserialize(cls, json_data):  
        params = json.loads(json_data)  
        return cls(*params['args'])
```

```
class BetterPoint2D(Deserializable):
    ...
before = BetterPoint2D(5, 3)
print('Before:    ', before)
data = before.serialize()
print('Serialized:', data)
after = BetterPoint2D.deserialize(data)
print('After:     ', after)

>>>
Before:      Point2D(5, 3)
Serialized: {"args": [5, 3]}
After:       Point2D(5, 3)
```



```
class BetterSerializable:
    def __init__(self, *args):
        self.args = args

    def serialize(self):
        return json.dumps({
            'class': self.__class__.__name__,
            'args': self.args,
        })

    def __repr__(self):
        name = self.__class__.__name__
        args_str = ', '.join(str(x) for x in self.args)
        return f'{name}({args_str})'
```

```
registry = {}

def register_class(target_class):
    registry[target_class.__name__] = target_class

def deserialize(data):
    params = json.loads(data)
    name = params['class']
    target_class = registry[name]
    return target_class(*params['args'])
```

```
class EvenBetterPoint2D(BetterSerializable):  
    def __init__(self, x, y):  
        super().__init__(x, y)  
        self.x = x  
        self.y = y  
  
register_class(EvenBetterPoint2D)
```

```
before = EvenBetterPoint2D(5, 3)
print('Before:      ', before)
data = before.serialize()
print('Serialized:', data)
after = deserialize(data)
print('After:       ', after)

>>>
Before:      EvenBetterPoint2D(5, 3)
Serialized: {"class": "EvenBetterPoint2D", "args": [5, 3]}
After:       EvenBetterPoint2D(5, 3)
```

```
class Point3D(BetterSerializable):  
    def __init__(self, x, y, z):  
        super().__init__(x, y, z)  
        self.x = x  
        self.y = y  
        self.z = z
```

# Forgot to call register\_class! Whoops!

```
class Meta(type):
    def __new__(meta, name, bases, class_dict):
        cls = type.__new__(meta, name, bases, class_dict)
        register_class(cls)
        return cls

class RegisteredSerializable(BetterSerializable,
                             metaclass=Meta):
    pass
```

```
class Vector3D(RegisteredSerializable):
    def __init__(self, x, y, z):
        super().__init__(x, y, z)
        self.x, self.y, self.z = x, y, z
```

```
before = Vector3D(10, -7, 3)
print('Before:      ', before)
data = before.serialize()
print('Serialized:', data)
print('After:       ', deserialize(data))
```

```
>>>
Before:      Vector3D(10, -7, 3)
Serialized: {"class": "Vector3D", "args": [10, -7, 3]}
After:       Vector3D(10, -7, 3)
```

```
class BetterRegisteredSerializable(BetterSerializable):
    def __init_subclass__(cls):
        super().__init_subclass__()
        register_class(cls)
```

```
class Vector1D(BetterRegisteredSerializable):
    def __init__(self, magnitude):
        super().__init__(magnitude)
        self.magnitude = magnitude
```

```
before = Vector1D(6)
print('Before: ', before)
data = before.serialize()
print('Serialized:', data)
print('After: ', deserialize(data))
```

```
>>>
Before:      Vector1D(6)
Serialized:  {"class": "Vector1D", "args": [6]}
After:       Vector1D(6)
```



```
class Field:
    def __init__(self, name):
        self.name = name
        self.internal_name = '_' + self.name

    def __get__(self, instance, instance_type):
        if instance is None:
            return self
        return getattr(instance, self.internal_name, '')

    def __set__(self, instance, value):
        setattr(instance, self.internal_name, value)
```

```
cust = Customer()
print(f'Before: {cust.first_name!r} {cust.__dict__}')
cust.first_name = 'Euclid'
print(f'After: {cust.first_name!r} {cust.__dict__}')

>>>
Before: '' {}
After: 'Euclid' {'_first_name': 'Euclid'}
```

```
class Customer:
    # Left side is redundant with right side
    first_name = Field('first_name')
    ...
```

```
class Meta(type):
    def __new__(meta, name, bases, class_dict):
        for key, value in class_dict.items():
            if isinstance(value, Field):
                value.name = key
                value.internal_name = '_' + key
        cls = type.__new__(meta, name, bases, class_dict)
        return cls
```

```
class Field:
    def __init__(self):
        # These will be assigned by the metaclass.
        self.name = None
        self.internal_name = None

    def __get__(self, instance, instance_type):
        if instance is None:
            return self
        return getattr(instance, self.internal_name, '')

    def __set__(self, instance, value):
        setattr(instance, self.internal_name, value)
```

```
cust = BetterCustomer()
print(f'Before: {cust.first_name!r} {cust.__dict__}')
cust.first_name = 'Euler'
print(f'After:  {cust.first_name!r} {cust.__dict__}')

>>>
Before: '' {}
After:  'Euler' {'_first_name': 'Euler'}
```

```
class BrokenCustomer:
    first_name = Field()
    last_name = Field()
    prefix = Field()
    suffix = Field()

cust = BrokenCustomer()
cust.first_name = 'Mersenne'

>>>
Traceback ...
TypeError: attribute name must be string, not 'NoneType'
```

```
class Field:
    def __init__(self):
        self.name = None
        self.internal_name = None

    def __set_name__(self, owner, name):
        # Called on class creation for each descriptor
        self.name = name
        self.internal_name = '_' + name

    def __get__(self, instance, instance_type):
        if instance is None:
            return self
        return getattr(instance, self.internal_name, '')

    def __set__(self, instance, value):
        setattr(instance, self.internal_name, value)
```



```
class FixedCustomer:
    first_name = Field()
    last_name = Field()
    prefix = Field()
    suffix = Field()

cust = FixedCustomer()
print(f'Before: {cust.first_name!r} {cust.__dict__}')
cust.first_name = 'Mersenne'
print(f'After: {cust.first_name!r} {cust.__dict__}')

>>>
Before: '' {}
After: 'Mersenne' {'_first_name': 'Mersenne'}
```

```

from functools import wraps

def trace_func(func):
    if hasattr(func, 'tracing'): # Only decorate once
        return func

    @wraps(func)
    def wrapper(*args, **kwargs):
        result = None
        try:
            result = func(*args, **kwargs)
            return result
        except Exception as e:
            result = e
            raise
        finally:
            print(f'{func.__name__}({args!r}, {kwargs!r}) -> '
                  f'{result!r}')

    wrapper.tracing = True
    return wrapper

```

```
class TraceDict(dict):
    @trace_func
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    @trace_func
    def __setitem__(self, *args, **kwargs):
        return super().__setitem__(*args, **kwargs)

    @trace_func
    def __getitem__(self, *args, **kwargs):
        return super().__getitem__(*args, **kwargs)

    ...
```

```
trace_dict = TraceDict([('hi', 1)])
trace_dict['there'] = 2
trace_dict['hi']
try:
    trace_dict['does not exist']
except KeyError:
    pass # Expected

>>>
__init__(({ 'hi': 1 }, [('hi', 1)]), {}) -> None
__setitem__(({ 'hi': 1, 'there': 2 }, 'there', 2), {}) -> None
__getitem__(({ 'hi': 1, 'there': 2 }, 'hi'), {}) -> 1
__getitem__(({ 'hi': 1, 'there': 2 }, 'does not exist'),
➡{}) -> KeyError('does not exist')
```

```
import types
```

```
trace_types = (  
    types.MethodType,  
    types.FunctionType,  
    types.BuiltinFunctionType,  
    types.BuiltinMethodType,  
    types.MethodDescriptorType,  
    types.ClassMethodDescriptorType)
```

```
class TraceMeta(type):  
    def __new__(meta, name, bases, class_dict):  
        klass = super().__new__(meta, name, bases, class_dict)  
  
        for key in dir(klass):  
            value = getattr(klass, key)  
            if isinstance(value, trace_types):  
                wrapped = trace_func(value)  
                setattr(klass, key, wrapped)  
  
        return klass
```

```

class TraceDict(dict, metaclass=TraceMeta):
    pass

trace_dict = TraceDict([('hi', 1)])
trace_dict['there'] = 2
trace_dict['hi']
try:
    trace_dict['does not exist']
except KeyError:
    pass # Expected

>>>
__new__((<class '__main__.TraceDict'>, [('hi', 1)]), {}) -> {}
__getitem__(({'hi': 1, 'there': 2}, 'hi'), {}) -> 1
__getitem__(({'hi': 1, 'there': 2}, 'does not exist'),
➡{}) -> KeyError('does not exist')

```

```
class OtherMeta(type):  
    pass
```

```
class SimpleDict(dict, metaclass=OtherMeta):  
    pass
```

```
class TraceDict(SimpleDict, metaclass=TraceMeta):  
    pass
```

```
>>>
```

```
Traceback ...
```

```
TypeError: metaclass conflict: the metaclass of a derived  
➡class must be a (non-strict) subclass of the metaclasses  
➡of all its bases
```

```

class TraceMeta(type):
    ...

class OtherMeta(TraceMeta):
    pass

class SimpleDict(dict, metaclass=OtherMeta):
    pass

class TraceDict(SimpleDict, metaclass=TraceMeta):
    pass

trace_dict = TraceDict([('hi', 1)])
trace_dict['there'] = 2
trace_dict['hi']
try:
    trace_dict['does not exist']
except KeyError:
    pass # Expected

>>>
__init_subclass__({}, {}) -> None
__new__((<class '__main__.TraceDict'>, [('hi', 1)]), {}) -> {}
__getitem__(({'hi': 1, 'there': 2}, 'hi'), {}) -> 1
__getitem__(({'hi': 1, 'there': 2}, 'does not exist'),
➡{}) -> KeyError('does not exist')

```



```
def trace(klass):  
    for key in dir(klass):  
        value = getattr(klass, key)  
        if isinstance(value, trace_types):  
            wrapped = trace_func(value)  
            setattr(klass, key, wrapped)  
    return klass
```

```

@trace
class TraceDict(dict):
    pass

trace_dict = TraceDict([('hi', 1)])
trace_dict['there'] = 2
trace_dict['hi']
try:
    trace_dict['does not exist']
except KeyError:
    pass # Expected

>>>
__new__((<class '__main__.TraceDict'>, [('hi', 1)]), {}) -> {}
__getitem__(({'hi': 1, 'there': 2}, 'hi'), {}) -> 1
__getitem__(({'hi': 1, 'there': 2}, 'does not exist'),
➡{}) -> KeyError('does not exist')

```



```
class OtherMeta(type):  
    pass
```

```
@trace  
class TraceDict(dict, metaclass=OtherMeta):  
    pass
```

```
trace_dict = TraceDict([('hi', 1)])  
trace_dict['there'] = 2  
trace_dict['hi']
```

```
try:  
    trace_dict['does not exist']  
except KeyError:  
    pass # Expected
```

```
>>>  
__new__((<class '__main__.TraceDict'>, [('hi', 1)]), {}) -> {}  
__getitem__(({'hi': 1, 'there': 2}, 'hi'), {}) -> 1  
__getitem__(({'hi': 1, 'there': 2}, 'does not exist'),  
➡{}) -> KeyError('does not exist')
```

```
import subprocess

result = subprocess.run(
    ['echo', 'Hello from the child!'],
    capture_output=True,
    encoding='utf-8')

result.check_returncode() # No exception means clean exit
print(result.stdout)

>>>
Hello from the child!
```

```
proc = subprocess.Popen(['sleep', '1'])
while proc.poll() is None:
    print('Working...')

    # Some time-consuming work here
    ...

print('Exit status', proc.poll())

>>>
Working...
Working...
Working...
Working...
Exit status 0
```

```
import time

start = time.time()
sleep_procs = []
for _ in range(10):
    proc = subprocess.Popen(['sleep', '1'])
    sleep_procs.append(proc)
```

```
for proc in sleep_procs:
    proc.communicate()

end = time.time()
delta = end - start
print(f'Finished in {delta:.3} seconds')

>>>
Finished in 1.05 seconds
```

```
import os
def run_encrypt(data):
    env = os.environ.copy()
    env['password'] = 'zf7ShyBhZ0raQDdE/FiZpm/m/8f9X+M1'
    proc = subprocess.Popen(
        ['openssl', 'enc', '-des3', '-pass', 'env:password'],
        env=env,
        stdin=subprocess.PIPE,
        stdout=subprocess.PIPE)
    proc.stdin.write(data)
    proc.stdin.flush() # Ensure that the child gets input
    return proc
```



```
def run_hash(input_stdin):  
    return subprocess.Popen(  
        ['openssl', 'dgst', '-whirlpool', '-binary'],  
        stdin=input_stdin,  
        stdout=subprocess.PIPE)
```

```
encrypt_procs = []
hash_procs = []
for _ in range(3):
    data = os.urandom(100)

    encrypt_proc = run_encrypt(data)
    encrypt_procs.append(encrypt_proc)

    hash_proc = run_hash(encrypt_proc.stdout)
    hash_procs.append(hash_proc)

    # Ensure that the child consumes the input stream and
    # the communicate() method doesn't inadvertently steal
    # input from the child. Also lets SIGPIPE propagate to
    # the upstream process if the downstream process dies.
    encrypt_proc.stdout.close()
    encrypt_proc.stdout = None
```

```
proc = subprocess.Popen(['sleep', '10'])
try:
    proc.communicate(timeout=0.1)
except subprocess.TimeoutExpired:
    proc.terminate()
    proc.wait()

print('Exit status', proc.poll())

>>>
Exit status -15
```

```
import time

numbers = [2139079, 1214759, 1516637, 1852285]
start = time.time()

for number in numbers:
    list(factorize(number))

end = time.time()
delta = end - start
print(f'Took {delta:.3f} seconds')

>>>
Took 0.399 seconds
```

```
from threading import Thread
```

```
class FactorizeThread(Thread):
```

```
    def __init__(self, number):
```

```
        super().__init__()
```

```
        self.number = number
```

```
    def run(self):
```

```
        self.factors = list(factorize(self.number))
```

```
start = time.time()
```

```
threads = []
```

```
for number in numbers:
```

```
    thread = FactorizeThread(number)
```

```
    thread.start()
```

```
    threads.append(thread)
```

```
import select
import socket

def slow_systemcall():
    select.select([socket.socket()], [], [], 0.1)
```

```
start = time.time()
threads = []
for _ in range(5):
    thread = Thread(target=slow_systemcall)
    thread.start()
    threads.append(thread)
```



```
def compute_helicopter_location(index):  
    ...  
  
for i in range(5):  
    compute_helicopter_location(i)  
  
for thread in threads:  
    thread.join()  
  
end = time.time()  
delta = end - start  
print(f'Took {delta:.3f} seconds')  
  
>>>  
Took 0.108 seconds
```

```
def worker(sensor_index, how_many, counter):  
    for _ in range(how_many):  
        # Read from the sensor  
        ...  
        counter.increment(1)
```

```
from threading import Thread

how_many = 10**5
counter = Counter()

threads = []
for i in range(5):
    thread = Thread(target=worker,
                    args=(i, how_many, counter))
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()

expected = how_many * 5
found = counter.count
print(f'Counter should be {expected}, got {found}')

>>>
Counter should be 500000, got 246760
```

```
counter = LockingCounter()

for i in range(5):
    thread = Thread(target=worker,
                    args=(i, how_many, counter))
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()

expected = how_many * 5
found = counter.count
print(f'Counter should be {expected}, got {found}')

>>>
Counter should be 500000, got 500000
```

```
from threading import Thread
import time
```

```
class Worker(Thread):
    def __init__(self, func, in_queue, out_queue):
        super().__init__()
        self.func = func
        self.in_queue = in_queue
        self.out_queue = out_queue
        self.polled_count = 0
        self.work_done = 0
```

```
def run(self):  
    while True:  
        self.polled_count += 1  
        try:  
            item = self.in_queue.get()  
        except IndexError:  
            time.sleep(0.01) # No work to do  
        else:  
            result = self.func(item)  
            self.out_queue.put(result)  
            self.work_done += 1
```

```
download_queue = MyQueue()
resize_queue = MyQueue()
upload_queue = MyQueue()
done_queue = MyQueue()
threads = [
    Worker(download, download_queue, resize_queue),
    Worker(resize, resize_queue, upload_queue),
    Worker(upload, upload_queue, done_queue),
]
```

```
while len(done_queue.items) < 1000:  
    # Do something useful while waiting  
    ...
```



```
processed = len(done_queue.items)
polled = sum(t.polled_count for t in threads)
print(f'Processed {processed} items after '
      f'polling {polled} times')
```

```
>>>
```

```
Processed 1000 items after polling 3035 times
```

```
from queue import Queue

my_queue = Queue()

def consumer():
    print('Consumer waiting')
    my_queue.get()          # Runs after put() below
    print('Consumer done')

thread = Thread(target=consumer)
thread.start()
```

```
print('Producer putting')
my_queue.put(object())
print('Producer done')
thread.join()
```

# Runs before get() above

```
>>>
```

```
Consumer waiting
Producer putting
Producer done
Consumer done
```

```
my_queue = Queue(1)                # Buffer size of 1

def consumer():
    time.sleep(0.1)                 # Wait
    my_queue.get()                  # Runs second
    print('Consumer got 1')
    my_queue.get()                  # Runs fourth
    print('Consumer got 2')
    print('Consumer done')

thread = Thread(target=consumer)
thread.start()
```

```
my_queue.put(object())           # Runs first
print('Producer put 1')
my_queue.put(object())           # Runs third
print('Producer put 2')
print('Producer done')
thread.join()
```

```
>>>
```

```
Producer put 1
Consumer got 1
Producer put 2
Producer done
Consumer got 2
Consumer done
```

```
in_queue = Queue()
def consumer():
    print('Consumer waiting')
    work = in_queue.get()          # Runs second
    print('Consumer working')
    # Doing work
    ...
    print('Consumer done')
    in_queue.task_done()          # Runs third

thread = Thread(target=consumer)
thread.start()
```

```
print('Producer putting')
in_queue.put(object())
print('Producer waiting')
in_queue.join()
print('Producer done')
thread.join()
```

# Runs first

# Runs fourth

```
>>>
```

```
Consumer waiting
Producer putting
Producer waiting
Consumer working
Consumer done
Producer done
```

```
def __iter__(self):
    while True:
        item = self.get()
        try:
            if item is self.SENTINEL:
                return # Cause the thread to exit
            yield item
        finally:
            self.task_done()
```



```
class StoppableWorker(Thread):  
    def __init__(self, func, in_queue, out_queue):  
        super().__init__()  
        self.func = func  
        self.in_queue = in_queue  
        self.out_queue = out_queue  
  
    def run(self):  
        for item in self.in_queue:  
            result = self.func(item)  
            self.out_queue.put(result)
```

```
download_queue = ClosableQueue()
resize_queue = ClosableQueue()
upload_queue = ClosableQueue()
done_queue = ClosableQueue()
threads = [
    StoppableWorker(download, download_queue, resize_queue),
    StoppableWorker(resize, resize_queue, upload_queue),
    StoppableWorker(upload, upload_queue, done_queue),
]
```

```
download_queue.join()
resize_queue.close()
resize_queue.join()
upload_queue.close()
upload_queue.join()
print(done_queue.qsize(), 'items finished')

for thread in threads:
    thread.join()

>>>
1000 items finished
```

```
def start_threads(count, *args):
    threads = [StoppableWorker(*args) for _ in range(count)]
    for thread in threads:
        thread.start()
    return threads

def stop_threads(closable_queue, threads):
    for _ in threads:
        closable_queue.close()

    closable_queue.join()

    for thread in threads:
        thread.join()
```

```
download_queue = ClosableQueue()
resize_queue = ClosableQueue()
upload_queue = ClosableQueue()
done_queue = ClosableQueue()

download_threads = start_threads(
    3, download, download_queue, resize_queue)
resize_threads = start_threads(
    4, resize, resize_queue, upload_queue)
upload_threads = start_threads(
    5, upload, upload_queue, done_queue)

for _ in range(1000):
    download_queue.put(object())

stop_threads(download_queue, download_threads)
stop_threads(resize_queue, resize_threads)
stop_threads(upload_queue, upload_threads)

print(done_queue.qsize(), 'items finished')

>>>
1000 items finished
```

0		1		2		3		4
-----		-----		-----		-----		-----
- *_ _ _ -		- _ * _ -		- _ ** _		- _ * _ -		- _ _ _ _
- _ ** _		- _ ** _		- * _ _ _		- * _ _ _		- ** _ _
- _ _ * _		- _ ** _		- _ ** _		- _ * _ -		- _ _ _ _
-----		-----		-----		-----		-----

```
class Grid:
    def __init__(self, height, width):
        self.height = height
        self.width = width
        self.rows = []
        for _ in range(self.height):
            self.rows.append([EMPTY] * self.width)
    def get(self, y, x):
        return self.rows[y % self.height][x % self.width]

    def set(self, y, x, state):
        self.rows[y % self.height][x % self.width] = state

    def __str__(self):
        ...
```

```
def count_neighbors(y, x, get):
    n_ = get(y - 1, x + 0) # North
    ne = get(y - 1, x + 1) # Northeast
    e_ = get(y + 0, x + 1) # East
    se = get(y + 1, x + 1) # Southeast
    s_ = get(y + 1, x + 0) # South
    sw = get(y + 1, x - 1) # Southwest
    w_ = get(y + 0, x - 1) # West
    nw = get(y - 1, x - 1) # Northwest
    neighbor_states = [n_, ne, e_, se, s_, sw, w_, nw]
    count = 0
    for state in neighbor_states:
        if state == ALIVE:
            count += 1
    return count
```



```
def game_logic(state, neighbors):  
    if state == ALIVE:  
        if neighbors < 2:  
            return EMPTY        # Die: Too few  
        elif neighbors > 3:  
            return EMPTY        # Die: Too many  
    else:  
        if neighbors == 3:  
            return ALIVE        # Regenerate  
    return state
```

```
def step_cell(y, x, get, set):  
    state = get(y, x)  
    neighbors = count_neighbors(y, x, get)  
    next_state = game_logic(state, neighbors)  
    set(y, x, next_state)
```

```
def simulate(grid):  
    next_grid = Grid(grid.height, grid.width)  
    for y in range(grid.height):  
        for x in range(grid.width):  
            step_cell(y, x, grid.get, next_grid.set)  
    return next_grid
```

```

class ColumnPrinter:
    ...

columns = ColumnPrinter()
for i in range(5):
    columns.append(str(grid))
    grid = simulate(grid)

print(columns)

```

```

>>>
      0      |      1      |      2      |      3      |      4
---*----- | ----- | ----- | ----- | -----
---*----- | --*-*----- | -----*----- | ---*----- | ----*-----
---*----- | --*-*----- | -----*----- | ---*----- | ----*-----
--***----- | ---**----- | --*-*----- | ---**----- | ----*-----
----- | ---*----- | ---**----- | ---**----- | ----***-----
----- | ----- | ----- | ----- | -----

```

```
def game_logic(state, neighbors):  
    ...  
    # Do some blocking input/output in here:  
    data = my_socket.recv(100)  
    ...
```

```
from threading import Lock
```

```
ALIVE = '*'
```

```
EMPTY = '-'
```

```
class Grid:
```

```
    ...
```

```
class LockingGrid(Grid):
```

```
    def __init__(self, height, width):  
        super().__init__(height, width)  
        self.lock = Lock()
```

```
    def __str__(self):  
        with self.lock:  
            return super().__str__()
```

```
    def get(self, y, x):  
        with self.lock:  
            return super().get(y, x)
```

```
    def set(self, y, x, state):  
        with self.lock:  
            return super().set(y, x, state)
```

```

from threading import Thread

def count_neighbors(y, x, get):
    ...

def game_logic(state, neighbors):
    ...
    # Do some blocking input/output in here:
    data = my_socket.recv(100)
    ...

def step_cell(y, x, get, set):
    state = get(y, x)
    neighbors = count_neighbors(y, x, get)
    next_state = game_logic(state, neighbors)
    set(y, x, next_state)

def simulate_threaded(grid):
    next_grid = LockingGrid(grid.height, grid.width)

    threads = []
    for y in range(grid.height):
        for x in range(grid.width):
            args = (y, x, grid.get, next_grid.set)
            thread = Thread(target=step_cell, args=args)
            thread.start() # Fan out
            threads.append(thread)

    for thread in threads:
        thread.join() # Fan in

    return next_grid

```

```

class ColumnPrinter:
    ...

grid = LockingGrid(5, 9) # Changed
grid.set(0, 3, ALIVE)
grid.set(1, 4, ALIVE)
grid.set(2, 2, ALIVE)
grid.set(2, 3, ALIVE)
grid.set(2, 4, ALIVE)

columns = ColumnPrinter()
for i in range(5):
    columns.append(str(grid))
    grid = simulate_threaded(grid) # Changed

```

```

print(columns)

```

```

>>>

```

0	1	2	3	4
---*-----	-----	-----	-----	-----
---*-----	--*_*-----	---*-----	---*-----	---*-----
--***-----	---**-----	--*_*-----	---**-----	---*-----
-----	---*-----	---**-----	---**-----	---***-----
-----	-----	-----	-----	-----



```
def game_logic(state, neighbors):  
    ...  
    raise OSError('Problem with I/O')  
    ...
```

```
import contextlib
import io
fake_stderr = io.StringIO()
with contextlib.redirect_stderr(fake_stderr):
    thread = Thread(target=game_logic, args=(ALIVE, 3))
    thread.start()
    thread.join()

print(fake_stderr.getvalue())

>>>
Exception in thread Thread-226:
Traceback (most recent call last):
  File "threading.py", line 917, in _bootstrap_inner
    self.run()
  File "threading.py", line 865, in run
    self._target(*self._args, **self._kwargs)
  File "example.py", line 193, in game_logic
    raise OSError('Problem with I/O')
OSError: Problem with I/O
```

```
from threading import Thread

class StoppableWorker(Thread):
    ...

def game_logic(state, neighbors):
    ...

    # Do some blocking input/output in here:
    data = my_socket.recv(100)
    ...

def game_logic_thread(item):
    y, x, state, neighbors = item
    try:
        next_state = game_logic(state, neighbors)
    except Exception as e:
        next_state = e
    return (y, x, next_state)

# Start the threads upfront
threads = []
for _ in range(5):
    thread = StoppableWorker(
        game_logic_thread, in_queue, out_queue)
    thread.start()
    threads.append(thread)
```

```
ALIVE = '*'
EMPTY = '-'
```

```
class SimulationError(Exception):
    pass
```

```
class Grid:
    ...
```

```
def count_neighbors(y, x, get):
    ...
```

```
def simulate_pipeline(grid, in_queue, out_queue):
    for y in range(grid.height):
        for x in range(grid.width):
            state = grid.get(y, x)
            neighbors = count_neighbors(y, x, grid.get)
            in_queue.put((y, x, state, neighbors)) # Fan out
```

```
    in_queue.join()
    out_queue.close()
```

```
    next_grid = Grid(grid.height, grid.width)
    for item in out_queue: # Fan in
        y, x, next_state = item
        if isinstance(next_state, Exception):
            raise SimulationError(y, x) from next_state
        next_grid.set(y, x, next_state)
```

```
    return next_grid
```

```
def game_logic(state, neighbors):  
    ...  
    raise OSError('Problem with I/O in game_logic')  
    ...
```

```
simulate_pipeline(Grid(1, 1), in_queue, out_queue)
```

```
>>>
```

```
Traceback ...
```

```
OSError: Problem with I/O in game_logic
```

The above exception was the direct cause of the following  
➡exception:

```
Traceback ...
```

```
SimulationError: (0, 0)
```

```

class ColumnPrinter:
    ...

grid = Grid(5, 9)
grid.set(0, 3, ALIVE)
grid.set(1, 4, ALIVE)
grid.set(2, 2, ALIVE)
grid.set(2, 3, ALIVE)
grid.set(2, 4, ALIVE)

columns = ColumnPrinter()
for i in range(5):
    columns.append(str(grid))
    grid = simulate_pipeline(grid, in_queue, out_queue)

print(columns)

for thread in threads:
    in_queue.close()
for thread in threads:
    thread.join()

```

```

>>>
      0      |      1      |      2      |      3      |      4
---*----- | ----- | ----- | ----- | -----
---*----- | ----- | --*-*----- | ----- | ---*-----
--***----- | ----- | ---**----- | ----- | --*-*-----
----- | ----- | ---*----- | ----- | ---**-----
----- | ----- | ----- | ----- | -----

```

```
def count_neighbors(y, x, get):  
    ...  
    # Do some blocking input/output in here:  
    data = my_socket.recv(100)  
    ...
```

```
def count_neighbors_thread(item):
    y, x, state, get = item
    try:
        neighbors = count_neighbors(y, x, get)
    except Exception as e:
        neighbors = e
    return (y, x, state, neighbors)

def game_logic_thread(item):
    y, x, state, neighbors = item
    if isinstance(neighbors, Exception):
        next_state = neighbors
    else:
        try:
            next_state = game_logic(state, neighbors)
        except Exception as e:
            next_state = e
    return (y, x, next_state)

class LockingGrid(Grid):
    ...
```



```
in_queue = ClosableQueue()
logic_queue = ClosableQueue()
out_queue = ClosableQueue()

threads = []

for _ in range(5):
    thread = StoppableWorker(
        count_neighbors_thread, in_queue, logic_queue)
    thread.start()
    threads.append(thread)

for _ in range(5):
    thread = StoppableWorker(
        game_logic_thread, logic_queue, out_queue)
    thread.start()
    threads.append(thread)
```

```

def simulate_phased_pipeline(
    grid, in_queue, logic_queue, out_queue):
    for y in range(grid.height):
        for x in range(grid.width):
            state = grid.get(y, x)
            item = (y, x, state, grid.get)
            in_queue.put(item)                # Fan out

    in_queue.join()
    logic_queue.join()                       # Pipeline sequencing
    out_queue.close()

    next_grid = LockingGrid(grid.height, grid.width)
    for item in out_queue:                   # Fan in
        y, x, next_state = item
        if isinstance(next_state, Exception):
            raise SimulationError(y, x) from next_state
        next_grid.set(y, x, next_state)

    return next_grid

```

```

grid = LockingGrid(5, 9)
grid.set(0, 3, ALIVE)
grid.set(1, 4, ALIVE)
grid.set(2, 2, ALIVE)
grid.set(2, 3, ALIVE)
grid.set(2, 4, ALIVE)
columns = ColumnPrinter()
for i in range(5):
    columns.append(str(grid))
    grid = simulate_phased_pipeline(
        grid, in_queue, logic_queue, out_queue)

```

```

print(columns)

```

```

for thread in threads:
    in_queue.close()
for thread in threads:
    logic_queue.close()
for thread in threads:
    thread.join()

```

```

>>>

```

0	1	2	3	4
---*-----	-----	-----	-----	-----
---*-----	--*_*----	---*-----	---*-----	---*-----
--***-----	---**-----	--*_*-----	---**-----	---**-----
-----	---*-----	---**-----	---**-----	---***-----
-----	-----	-----	-----	-----

```
ALIVE = '*'
```

```
EMPTY = '-'
```

```
class Grid:
```

```
    ...
```

```
class LockingGrid(Grid):
```

```
    ...
```

```
def count_neighbors(y, x, get):
```

```
    ...
```

```
def game_logic(state, neighbors):
```

```
    ...
```

```
    # Do some blocking input/output in here:
```

```
    data = my_socket.recv(100)
```

```
    ...
```

```
def step_cell(y, x, get, set):
```

```
    state = get(y, x)
```

```
    neighbors = count_neighbors(y, x, get)
```

```
    next_state = game_logic(state, neighbors)
```

```
    set(y, x, next_state)
```

```
from concurrent.futures import ThreadPoolExecutor

def simulate_pool(pool, grid):
    next_grid = LockingGrid(grid.height, grid.width)
    futures = []
    for y in range(grid.height):
        for x in range(grid.width):
            args = (y, x, grid.get, next_grid.set)
            future = pool.submit(step_cell, *args) # Fan out
            futures.append(future)

    for future in futures:
        future.result() # Fan in

    return next_grid
```

```

class ColumnPrinter:
    ...

grid = LockingGrid(5, 9)
grid.set(0, 3, ALIVE)
grid.set(1, 4, ALIVE)
grid.set(2, 2, ALIVE)
grid.set(2, 3, ALIVE)
grid.set(2, 4, ALIVE)

columns = ColumnPrinter()
with ThreadPoolExecutor(max_workers=10) as pool:
    for i in range(5):
        columns.append(str(grid))
        grid = simulate_pool(pool, grid)

print(columns)

```

```

>>>
      0      |      1      |      2      |      3      |      4
----*-----|-----*-----|-----*-----|-----*-----|-----*-----
----*-----|----*-*------|-----*-----|----*-*------|-----*-----
--***-----|---**-----|---*-*------|---**-----|-----*------
-----*-----|---*------|---**-----|---**-----|---***-----
-----*-----|-----*-----|-----*-----|-----*-----|-----*-----

```

```
def game_logic(state, neighbors):
    ...
    raise OSError('Problem with I/O')
    ...

with ThreadPoolExecutor(max_workers=10) as pool:
    task = pool.submit(game_logic, ALIVE, 3)
    task.result()

>>>
Traceback ...
OSError: Problem with I/O
```

```
ALIVE = '*'
EMPTY = '-'

class Grid:
    ...

def count_neighbors(y, x, get):
    ...

async def game_logic(state, neighbors):
    ...
    # Do some input/output in here:
    data = await my_socket.read(50)
    ...
```



```
async def step_cell(y, x, get, set):  
    state = get(y, x)  
    neighbors = count_neighbors(y, x, get)  
    next_state = await game_logic(state, neighbors)  
    set(y, x, next_state)
```

```
import asyncio

async def simulate(grid):
    next_grid = Grid(grid.height, grid.width)

    tasks = []
    for y in range(grid.height):
        for x in range(grid.width):
            task = step_cell(
                y, x, grid.get, next_grid.set)      # Fan out
            tasks.append(task)

    await asyncio.gather(*tasks)                  # Fan in

    return next_grid
```

```

class ColumnPrinter:
    ...

grid = Grid(5, 9)
grid.set(0, 3, ALIVE)
grid.set(1, 4, ALIVE)
grid.set(2, 2, ALIVE)
grid.set(2, 3, ALIVE)
grid.set(2, 4, ALIVE)

columns = ColumnPrinter()
for i in range(5):
    columns.append(str(grid))
    grid = asyncio.run(simulate(grid))    # Run the event loop

print(columns)

```

```

>>>
      0      |      1      |      2      |      3      |      4
----*----- | ----*----- | ----*----- | ----*----- | ----*-----
----*----- | --*_*----- | ----*----- | ----*----- | ----*-----
--***----- | ---**----- | --*_*----- | ----**----- | ----*-----
-----      | ---*----- | ---**----- | ----**----- | ----***-----
-----      | -----      | -----      | -----      | -----

```

```
async def game_logic(state, neighbors):  
    ...  
    raise OSError('Problem with I/O')  
    ...
```

```
asyncio.run(game_logic(ALIVE, 3))
```

```
>>>
```

```
Traceback ...
```

```
OSError: Problem with I/O
```

```
async def count_neighbors(y, x, get):
```

```
    ...
```

```
async def step_cell(y, x, get, set):
```

```
    state = get(y, x)
```

```
    neighbors = await count_neighbors(y, x, get)
```

```
    next_state = await game_logic(state, neighbors)
```

```
    set(y, x, next_state)
```

```
grid = Grid(5, 9)
```

```
grid.set(0, 3, ALIVE)
```

```
grid.set(1, 4, ALIVE)
```

```
grid.set(2, 2, ALIVE)
```

```
grid.set(2, 3, ALIVE)
```

```
grid.set(2, 4, ALIVE)
```

```
columns = ColumnPrinter()
```

```
for i in range(5):
```

```
    columns.append(str(grid))
```

```
    grid = asyncio.run(simulate(grid))
```

```
print(columns)
```

```
>>>
```

0	1	2	3	4
---*-----	-----	-----	-----	-----
---*-----	--*_*-----	---*-----	---*-----	---*-----
--***-----	---**-----	--*_*-----	---**-----	---*-----
-----	---*-----	---**-----	---**-----	---***-----
-----	-----	-----	-----	-----

```
class EOFError(Exception):  
    pass
```

```
class ConnectionBase:  
    def __init__(self, connection):  
        self.connection = connection  
        self.file = connection.makefile('rb')  
  
    def send(self, command):  
        line = command + '\n'  
        data = line.encode()  
        self.connection.send(data)  
  
    def receive(self):  
        line = self.file.readline()  
        if not line:  
            raise EOFError('Connection closed')  
        return line[:-1].decode()
```

```
import random
```

```
WARMER = 'Warmer'
```

```
COLDER = 'Colder'
```

```
UNSURE = 'Unsure'
```

```
CORRECT = 'Correct'
```

```
class UnknownCommandError(Exception):  
    pass
```

```
class Session(ConnectionBase):  
    def __init__(self, *args):  
        super().__init__(*args)  
        self._clear_state(None, None)  
  
    def _clear_state(self, lower, upper):  
        self.lower = lower  
        self.upper = upper  
        self.secret = None  
        self.guesses = []
```

```
def loop(self):
    while command := self.receive():
        parts = command.split(' ')
        if parts[0] == 'PARAMS':
            self.set_params(parts)
        elif parts[0] == 'NUMBER':
            self.send_number()
        elif parts[0] == 'REPORT':
            self.receive_report(parts)
        else:
            raise UnknownCommandError(command)
```



```
def set_params(self, parts):  
    assert len(parts) == 3  
    lower = int(parts[1])  
    upper = int(parts[2])  
    self._clear_state(lower, upper)
```

```
def next_guess(self):  
    if self.secret is not None:  
        return self.secret  
  
    while True:  
        guess = random.randint(self.lower, self.upper)  
        if guess not in self.guesses:  
            return guess  
  
def send_number(self):  
    guess = self.next_guess()  
    self.guesses.append(guess)  
    self.send(format(guess))
```

```
def receive_report(self, parts):  
    assert len(parts) == 2  
    decision = parts[1]  
  
    last = self.guesses[-1]  
    if decision == CORRECT:  
        self.secret = last  
  
    print(f'Server: {last} is {decision}')
```

```
@contextlib.contextmanager
def session(self, lower, upper, secret):
    print(f'Guess a number between {lower} and {upper}!'
          f' Shhhhh, it\'s {secret}.')
    self.secret = secret
    self.send(f'PARAMS {lower} {upper}')
    try:
        yield
    finally:
        self._clear_state()
        self.send('PARAMS 0 -1')
```

```
def request_numbers(self, count):  
    for _ in range(count):  
        self.send('NUMBER')  
        data = self.receive()  
        yield int(data)  
    if self.last_distance == 0:  
        return
```

```
def report_outcome(self, number):  
    new_distance = math.fabs(number - self.secret)  
    decision = UNSURE  
  
    if new_distance == 0:  
        decision = CORRECT  
    elif self.last_distance is None:  
        pass  
    elif new_distance < self.last_distance:  
        decision = WARMER  
    elif new_distance > self.last_distance:  
        decision = COLDER  
  
    self.last_distance = new_distance  
  
    self.send(f'REPORT {decision}')  
    return decision
```

```
import socket
from threading import Thread

def handle_connection(connection):
    with connection:
        session = Session(connection)
        try:
            session.loop()
        except EOFError:
            pass

def run_server(address):
    with socket.socket() as listener:
        listener.bind(address)
        listener.listen()
        while True:
            connection, _ = listener.accept()
            thread = Thread(target=handle_connection,
                            args=(connection,),
                            daemon=True)
            thread.start()
```

```
def run_client(address):  
    with socket.create_connection(address) as connection:  
        client = Client(connection)  
  
        with client.session(1, 5, 3):  
            results = [(x, client.report_outcome(x))  
                        for x in client.request_numbers(5)]  
  
        with client.session(10, 15, 12):  
            for number in client.request_numbers(5):  
                outcome = client.report_outcome(number)  
                results.append((number, outcome))  
  
    return results
```



```
def main():
    address = ('127.0.0.1', 1234)
    server_thread = Thread(
        target=run_server, args=(address,), daemon=True)
    server_thread.start()

    results = run_client(address)
    for number, outcome in results:
        print(f'Client: {number} is {outcome}')
```

```
main()
```

```
>>>
```

```
Guess a number between 1 and 5! Shhhhh, it's 3.
```

```
Server: 4 is Unsure
```

```
Server: 1 is Colder
```

```
Server: 5 is Unsure
```

```
Server: 3 is Correct
```

```
Guess a number between 10 and 15! Shhhhh, it's 12.
```

```
Server: 11 is Unsure
```

```
Server: 10 is Colder
```

```
Server: 12 is Correct
```

```
Client: 4 is Unsure
```

```
Client: 1 is Colder
```

```
Client: 5 is Unsure
```

```
Client: 3 is Correct
```

```
Client: 11 is Unsure
```

```
Client: 10 is Colder
```

```
Client: 12 is Correct
```

```
class AsyncConnectionBase:
    def __init__(self, reader, writer):
        self.reader = reader
        self.writer = writer

    async def send(self, command):
        line = command + '\n'
        data = line.encode()
        self.writer.write(data)
        await self.writer.drain()

    async def receive(self):
        line = await self.reader.readline()
        if not line:
            raise EOFError('Connection closed')
        return line[:-1].decode()
```

```
class AsyncSession(AsyncConnectionBase):           # Changed
    def __init__(self, *args):
        ...

    def _clear_values(self, lower, upper):
        ...
```

```
async def loop(self):                                # Changed
while command := await self.receive():                # Changed
    parts = command.split(' ')
    if parts[0] == 'PARAMS':
        self.set_params(parts)
    elif parts[0] == 'NUMBER':
        await self.send_number()                      # Changed
    elif parts[0] == 'REPORT':
        self.receive_report(parts)
    else:
        raise UnknownCommandError(command)
```

```
def next_guess(self):
```

```
    ...
```

```
async def send_number(self):
```

```
# Changed
```

```
    guess = self.next_guess()
```

```
    self.guesses.append(guess)
```

```
    await self.send(format(guess))
```

```
# Changed
```

```
class AsyncClient(AsyncConnectionBase):           # Changed
    def __init__(self, *args):
        ...

    def _clear_state(self):
        ...
```

```
@contextlib.asynccontextmanager                                # Changed
async def session(self, lower, upper, secret):                 # Changed
    print(f'Guess a number between {lower} and {upper}!'      # Changed
          f' Shhhhh, it\'s {secret}.')
    self.secret = secret
    await self.send(f'PARAMS {lower} {upper}')                 # Changed
    try:
        yield
    finally:
        self._clear_state()
        await self.send('PARAMS 0 -1')                          # Changed
```

```
async def request_numbers(self, count):           # Changed
    for _ in range(count):
        await self.send('NUMBER')                # Changed
        data = await self.receive()              # Changed
        yield int(data)
    if self.last_distance == 0:
        return
```



```
async def report_outcome(self, number):          # Changed
    ...
    await self.send(f'REPORT {decision}')        # Changed
    ...
```

```
import asyncio

async def handle_async_connection(reader, writer):
    session = AsyncSession(reader, writer)
    try:
        await session.loop()
    except EOFError:
        pass

async def run_async_server(address):
    server = await asyncio.start_server(
        handle_async_connection, *address)
    async with server:
        await server.serve_forever()
```

```

async def run_async_client(address):
    streams = await asyncio.open_connection(*address)    # New
    client = AsyncClient(*streams)                      # New

    async with client.session(1, 5, 3):
        results = [(x, await client.report_outcome(x))
                    async for x in client.request_numbers(5)]

    async with client.session(10, 15, 12):
        async for number in client.request_numbers(5):
            outcome = await client.report_outcome(number)
            results.append((number, outcome))

    _, writer = streams                                # New
    writer.close()                                     # New
    await writer.wait_closed()                         # New

    return results

```

```
async def main_async():
    address = ('127.0.0.1', 4321)

    server = run_async_server(address)
    asyncio.create_task(server)

    results = await run_async_client(address)
    for number, outcome in results:
        print(f'Client: {number} is {outcome}')

asyncio.run(main_async())
```

>>>

Guess a number between 1 and 5! Shhhhh, it's 3.

Server: 5 is Unsure

Server: 4 is Warmer

Server: 2 is Unsure

Server: 1 is Colder

Server: 3 is Correct

Guess a number between 10 and 15! Shhhhh, it's 12.

Server: 14 is Unsure

Server: 10 is Unsure

Server: 15 is Colder

Server: 12 is Correct

Client: 5 is Unsure

Client: 4 is Warmer

Client: 2 is Unsure

Client: 1 is Colder

Client: 3 is Correct

Client: 14 is Unsure

Client: 10 is Unsure

Client: 15 is Colder

Client: 12 is Correct

```
import time

def tail_file(handle, interval, write_func):
    while not handle.closed:
        try:
            line = readline(handle)
        except NoNewData:
            time.sleep(interval)
        else:
            write_func(line)
```

```
from threading import Lock, Thread

def run_threads(handles, interval, output_path):
    with open(output_path, 'wb') as output:
        lock = Lock()
        def write(data):
            with lock:
                output.write(data)
    threads = []

    for handle in handles:
        args = (handle, interval, write)
        thread = Thread(target=tail_file, args=args)
        thread.start()
        threads.append(thread)

    for thread in threads:
        thread.join()
```

```
def confirm_merge(input_paths, output_path):  
    ...  
  
    input_paths = ...  
    handles = ...  
    output_path = ...  
    run_threads(handles, 0.1, output_path)  
  
confirm_merge(input_paths, output_path)
```



```
import asyncio

async def run_tasks_mixed(handles, interval, output_path):
    loop = asyncio.get_event_loop()

    with open(output_path, 'wb') as output:
        async def write_async(data):
            output.write(data)

        def write(data):
            coro = write_async(data)
            future = asyncio.run_coroutine_threadsafe(
                coro, loop)
            future.result()

        tasks = []
        for handle in handles:
            task = loop.run_in_executor(
                None, tail_file, handle, interval, write)
            tasks.append(task)

    await asyncio.gather(*tasks)
```

```
input_paths = ...
handles = ...
output_path = ...
asyncio.run(run_tasks_mixed(handles, 0.1, output_path))

confirm_merge(input_paths, output_path)
```

```
async def tail_async(handle, interval, write_func):
    loop = asyncio.get_event_loop()

    while not handle.closed:
        try:
            line = await loop.run_in_executor(
                None, readline, handle)
        except NoNewData:
            await asyncio.sleep(interval)
        else:
            await write_func(line)
```

```
async def run_tasks(handles, interval, output_path):
    with open(output_path, 'wb') as output:
        async def write_async(data):
            output.write(data)

        tasks = []
        for handle in handles:
            coro = tail_async(handle, interval, write_async)
            task = asyncio.create_task(coro)
            tasks.append(task)

        await asyncio.gather(*tasks)
```

```
input_paths = ...  
handles = ...  
output_path = ...  
asyncio.run(run_tasks(handles, 0.1, output_path))  
  
confirm_merge(input_paths, output_path)
```

```
def tail_file(handle, interval, write_func):  
    loop = asyncio.new_event_loop()  
    asyncio.set_event_loop(loop)  
  
    async def write_async(data):  
        write_func(data)  
  
    coro = tail_async(handle, interval, write_async)  
    loop.run_until_complete(coro)
```

```
input_paths = ...  
handles = ...  
output_path = ...  
run_threads(handles, 0.1, output_path)  
  
confirm_merge(input_paths, output_path)
```

```
import asyncio

async def run_tasks(handles, interval, output_path):
    with open(output_path, 'wb') as output:
        async def write_async(data):
            output.write(data)

        tasks = []
        for handle in handles:
            coro = tail_async(handle, interval, write_async)
            task = asyncio.create_task(coro)
            tasks.append(task)

        await asyncio.gather(*tasks)
```



```
import time

async def slow_coroutine():
    time.sleep(0.5) # Simulating slow I/O

asyncio.run(slow_coroutine(), debug=True)

>>>
Executing <Task finished name='Task-1' coro=<slow_coroutine()
➡done, defined at example.py:29> result=None created
➡at .../asyncio/base_events.py:487> took 0.503 seconds
...
```

```
from threading import Thread
```

```
class WriteThread(Thread):
```

```
    def __init__(self, output_path):
```

```
        super().__init__()
```

```
        self.output_path = output_path
```

```
        self.output = None
```

```
        self.loop = asyncio.new_event_loop()
```

```
    def run(self):
```

```
        asyncio.set_event_loop(self.loop)
```

```
        with open(self.output_path, 'wb') as self.output:
```

```
            self.loop.run_forever()
```

```
        # Run one final round of callbacks so the await on
```

```
        # stop() in another event loop will be resolved.
```

```
        self.loop.run_until_complete(asyncio.sleep(0))
```

```
async def real_write(self, data):  
    self.output.write(data)
```

```
async def write(self, data):  
    coro = self.real_write(data)  
    future = asyncio.run_coroutine_threadsafe(  
        coro, self.loop)  
    await asyncio.wrap_future(future)
```

```
async def real_stop(self):  
    self.loop.stop()  
async def stop(self):  
    coro = self.real_stop()  
    future = asyncio.run_coroutine_threadsafe(  
        coro, self.loop)  
    await asyncio.wrap_future(future)
```

```
async def __aenter__(self):  
    loop = asyncio.get_event_loop()  
    await loop.run_in_executor(None, self.start)  
    return self  
  
async def __aexit__(self, *_):  
    await self.stop()
```

```
def readline(handle):
    ...

async def tail_async(handle, interval, write_func):
    ...

async def run_fully_async(handles, interval, output_path):
    async with WriteThread(output_path) as output:
        tasks = []
        for handle in handles:
            coro = tail_async(handle, interval, output.write)
            task = asyncio.create_task(coro)
            tasks.append(task)

        await asyncio.gather(*tasks)
```

```
def confirm_merge(input_paths, output_path):  
    ...  
    input_paths = ...  
    handles = ...  
    output_path = ...  
    asyncio.run(run_fully_async(handles, 0.1, output_path))  
  
confirm_merge(input_paths, output_path)
```

```
# my_module.py
def gcd(pair):
    a, b = pair
    low = min(a, b)
    for i in range(low, 0, -1):
        if a % i == 0 and b % i == 0:
            return i
    assert False, 'Not reachable'
```



```
# run_serial.py
import my_module
import time

NUMBERS = [
    (1963309, 2265973), (2030677, 3814172),
    (1551645, 2229620), (2039045, 2020802),
    (1823712, 1924928), (2293129, 1020491),
    (1281238, 2273782), (3823812, 4237281),
    (3812741, 4729139), (1292391, 2123811),
]

def main():
    start = time.time()
    results = list(map(my_module.gcd, NUMBERS))
    end = time.time()
    delta = end - start
    print(f'Took {delta:.3f} seconds')

if __name__ == '__main__':
    main()

>>>
Took 1.173 seconds
```

```
# run_threads.py
import my_module
from concurrent.futures import ThreadPoolExecutor
import time

NUMBERS = [
    ...
]

def main():
    start = time.time()
    pool = ThreadPoolExecutor(max_workers=2)
    results = list(pool.map(my_module.gcd, NUMBERS))
    end = time.time()
    delta = end - start
    print(f'Took {delta:.3f} seconds')

if __name__ == '__main__':
    main()

>>>
Took 1.436 seconds
```

```
# run_parallel.py
import my_module
from concurrent.futures import ProcessPoolExecutor
import time

NUMBERS = [
    ...
]

def main():
    start = time.time()
    pool = ProcessPoolExecutor(max_workers=2) # The one change
    results = list(pool.map(my_module.gcd, NUMBERS))
    end = time.time()
    delta = end - start
    print(f'Took {delta:.3f} seconds')

if __name__ == '__main__':
    main()

>>>
Took 0.683 seconds
```

```
def try_finally_example(filename):  
    print('* Opening file')  
    handle = open(filename, encoding='utf-8') # Maybe OSError  
    try:  
        print('* Reading data')  
        return handle.read() # Maybe UnicodeDecodeError  
    finally:  
        print('* Calling close()')  
        handle.close() # Always runs after try block
```

```
filename = 'random_data.txt'
```

```
with open(filename, 'wb') as f:
```

```
    f.write(b'\xf1\xf2\xf3\xf4\xf5') # Invalid utf-8
```

```
data = try_finally_example(filename)
```

```
>>>
```

```
* Opening file
```

```
* Reading data
```

```
* Calling close()
```

```
Traceback ...
```

```
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xf1 in
```

```
➡position 0: invalid continuation byte
```

```
try_finally_example('does_not_exist.txt')
```

```
>>>
```

```
* Opening file
```

```
Traceback ...
```

```
FileNotFoundError: [Errno 2] No such file or directory:
```

```
↳ 'does_not_exist.txt'
```

```
import json

def load_json_key(data, key):
    try:
        print('* Loading JSON data')
        result_dict = json.loads(data) # May raise ValueError
    except ValueError as e:
        print('* Handling ValueError')
        raise KeyError(key) from e
    else:
        print('* Looking up key')
        return result_dict[key] # May raise KeyError
```

```
assert load_json_key('{"foo": "bar"}', 'foo') == 'bar'
```

```
>>>
```

```
* Loading JSON data
```

```
* Looking up key
```



```
load_json_key('{"foo": bad payload', 'foo')
```

```
>>>
```

```
* Loading JSON data
```

```
* Handling ValueError
```

```
Traceback ...
```

```
JSONDecodeError: Expecting value: line 1 column 9 (char 8)
```

The above exception was the direct cause of the following  
➡exception:

```
Traceback ...
```

```
KeyError: 'foo'
```

```
load_json_key({'foo': 'bar'}, 'does not exist')
>>>
* Loading JSON data
* Looking up key
Traceback ...
KeyError: 'does not exist'
```

```
UNDEFINED = object()
```

```
def divide_json(path):  
    print('* Opening file')  
    handle = open(path, 'r+')    # May raise OSError  
    try:  
        print('* Reading data')  
        data = handle.read()    # May raise UnicodeDecodeError  
        print('* Loading JSON data')  
        op = json.loads(data)    # May raise ValueError  
        print('* Performing calculation')  
        value = (  
            op['numerator'] /  
            op['denominator'])    # May raise ZeroDivisionError  
    except ZeroDivisionError as e:  
        print('* Handling ZeroDivisionError')  
        return UNDEFINED  
    else:  
        print('* Writing calculation')  
        op['result'] = value  
        result = json.dumps(op)  
        handle.seek(0)          # May raise OSError  
        handle.write(result)    # May raise OSError  
        return value  
    finally:  
        print('* Calling close()')  
        handle.close()          # Always runs
```

```
temp_path = 'random_data.json'

with open(temp_path, 'w') as f:
    f.write('{"numerator": 1, "denominator": 10}')

assert divide_json(temp_path) == 0.1

>>>
* Opening file
* Reading data
* Loading JSON data
* Performing calculation
* Writing calculation
* Calling close()
```

```
with open(temp_path, 'w') as f:  
    f.write('{"numerator": 1, "denominator": 0}')
```

```
assert divide_json(temp_path) is UNDEFINED
```

```
>>>
```

```
* Opening file  
* Reading data  
* Loading JSON data  
* Performing calculation  
* Handling ZeroDivisionError  
* Calling close()
```

```
with open(temp_path, 'w') as f:  
    f.write('{"numerator": 1 bad data'})
```

```
divide_json(temp_path)
```

```
>>>
```

```
* Opening file
```

```
* Reading data
```

```
* Loading JSON data
```

```
* Calling close()
```

```
Traceback ...
```

```
JSONDecodeError: Expecting ',' delimiter: line 1 column 17
```

```
➡(char 16)
```

```
with open(temp_path, 'w') as f:  
    f.write('{"numerator": 1, "denominator": 10}')
```

```
divide_json(temp_path)
```

```
>>>
```

```
* Opening file  
* Reading data  
* Loading JSON data  
* Performing calculation  
* Writing calculation  
* Calling close()
```

```
Traceback ...
```

```
OSError: [Errno 28] No space left on device
```

```
from threading import Lock

lock = Lock()
with lock:
    # Do something while maintaining an invariant
    ...
```



```
lock.acquire()
try:
    # Do something while maintaining an invariant
    ...
finally:
    lock.release()
```

```
import logging
```

```
def my_function():  
    logging.debug('Some debug data')  
    logging.error('Error log here')  
    logging.debug('More debug data')
```

```
from contextlib import contextmanager

@contextmanager
def debug_logging(level):
    logger = logging.getLogger()
    old_level = logger.getEffectiveLevel()
    logger.setLevel(level)
    try:
        yield
    finally:
        logger.setLevel(old_level)
```

```
with open('my_output.txt', 'w') as handle:  
    handle.write('This is some data!')
```

```
@contextmanager
def log_level(level, name):
    logger = logging.getLogger(name)
    old_level = logger.getEffectiveLevel()
    logger.setLevel(level)
    try:
        yield logger
    finally:
        logger.setLevel(old_level)
```

```
with log_level(logging.DEBUG, 'my-log') as logger:  
    logger.debug(f'This is a message for {logger.name}!')  
    logging.debug('This will not print')
```

```
>>>
```

```
This is a message for my-log!
```

```
logger = logging.getLogger('my-log')  
logger.debug('Debug will not print')  
logger.error('Error will print')
```

```
>>>
```

```
Error will print
```

```
with log_level(logging.DEBUG, 'other-log') as logger:  
    logger.debug(f'This is a message for {logger.name}!')  
    logging.debug('This will not print')
```

```
>>>
```

```
This is a message for other-log!
```



```
import time

now = 1552774475
local_tuple = time.localtime(now)
time_format = '%Y-%m-%d %H:%M:%S'
time_str = time.strftime(time_format, local_tuple)
print(time_str)

>>>
2019-03-16 15:14:35
```

```
time_tuple = time.strptime(time_str, time_format)
utc_now = time.mktime(time_tuple)
print(utc_now)
```

```
>>>
```

```
1552774475.0
```

```
import os

if os.name == 'nt':
    print("This example doesn't work on Windows")
else:
    parse_format = '%Y-%m-%d %H:%M:%S %Z'
    depart_sfo = '2019-03-16 15:45:16 PDT'
    time_tuple = time.strptime(depart_sfo, parse_format)
    time_str = time.strftime(time_format, time_tuple)
    print(time_str)

>>>
2019-03-16 15:45:16
```

```
arrival_nyc = '2019-03-16 23:33:24 EDT'  
time_tuple = time.strptime(arrival_nyc, time_format)
```

```
>>>
```

```
Traceback ...
```

```
ValueError: unconverted data remains:  EDT
```

```
from datetime import datetime, timezone

now = datetime(2019, 3, 16, 22, 14, 35)
now_utc = now.replace(tzinfo=timezone.utc)
now_local = now_utc.astimezone()
print(now_local)

>>>
2019-03-16 15:14:35-07:00
```

```
time_str = '2019-03-16 15:14:35'  
now = datetime.strptime(time_str, time_format)  
time_tuple = now.timetuple()  
utc_now = time.mktime(time_tuple)  
print(utc_now)
```

```
>>>
```

```
1552774475.0
```

```
import pytz

arrival_nyc = '2019-03-16 23:33:24'
nyc_dt_naive = datetime.strptime(arrival_nyc, time_format)
eastern = pytz.timezone('US/Eastern')
nyc_dt = eastern.localize(nyc_dt_naive)
utc_dt = pytz.utc.normalize(nyc_dt.astimezone(pytz.utc))
print(utc_dt)

>>>
2019-03-17 03:33:24+00:00
```

```
pacific = pytz.timezone('US/Pacific')  
sf_dt = pacific.normalize(utc_dt.astimezone(pacific))  
print(sf_dt)
```

```
>>>
```

```
2019-03-16 20:33:24-07:00
```



```
nepal = pytz.timezone('Asia/Katmandu')  
nepal_dt = nepal.normalize(utc_dt.astimezone(nepal))  
print(nepal_dt)
```

```
>>>
```

```
2019-03-17 09:18:24+05:45
```

```
state = GameState()
state.level += 1 # Player beat a level
state.lives -= 1 # Player had to try again

print(state.__dict__)

>>>
{'level': 1, 'lives': 3}
```

```
state = GameState()
serialized = pickle.dumps(state)
state_after = pickle.loads(serialized)
print(state_after.__dict__)
```

```
>>>
```

```
{'level': 0, 'lives': 4, 'points': 0}
```

```
assert isinstance(state_after, GameState)
```

```
class GameState:  
    def __init__(self, level=0, lives=4, points=0):  
        self.level = level  
        self.lives = lives  
        self.points = points
```

```
def pickle_game_state(game_state):  
    kwargs = game_state.__dict__  
    return unpickle_game_state, (kwargs,)
```

```
import copyreg
```

```
copyreg.pickle(GameState, pickle_game_state)
```

```
state = GameState()
state.points += 1000
serialized = pickle.dumps(state)
state_after = pickle.loads(serialized)
print(state_after.__dict__)

>>>
{'level': 0, 'lives': 4, 'points': 1000}
```



```
class GameState:
    def __init__(self, level=0, lives=4, points=0, magic=5):
        self.level = level
        self.lives = lives
        self.points = points
        self.magic = magic # New field
```

```
print('Before:', state.__dict__)
state_after = pickle.loads(serialized)
print('After: ', state_after.__dict__)

>>>
Before: {'level': 0, 'lives': 4, 'points': 1000}
After:  {'level': 0, 'lives': 4, 'points': 1000, 'magic': 5}
```

```
class GameState:
    def __init__(self, level=0, points=0, magic=5):
        self.level = level
        self.points = points
        self.magic = magic
```

```
pickle.loads(serialized)
```

```
>>>
```

```
Traceback ...
```

```
TypeError: __init__() got an unexpected keyword argument
```

```
↳ 'lives'
```

```
copyreg.pickle(GameState, pickle_game_state)
print('Before:', state.__dict__)
state_after = pickle.loads(serialized)
print('After: ', state_after.__dict__)

>>>
Before: {'level': 0, 'lives': 4, 'points': 1000}
After:  {'level': 0, 'points': 1000, 'magic': 5}
```

```
class BetterGameState:
    def __init__(self, level=0, points=0, magic=5):
        self.level = level
        self.points = points
        self.magic = magic
```

```
pickle.loads(serialized)
```

```
>>>
```

```
Traceback ...
```

```
AttributeError: Can't get attribute 'GameState' on <module
```

```
↳ '__main__' from 'my_code.py'>
```

```
print(serialized)
```

```
>>>
```

```
b'\x80\x04\x95A\x00\x00\x00\x00\x00\x00\x00\x8c\x08__main__  
➡\x94\x8c\tGameState\x94\x93\x94)\x81\x94}\x94(\x8c\x05level  
➡\x94K\x00\x8c\x06points\x94K\x00\x8c\x05magic\x94K\x05ub.'
```



```
copyreg.pickle(BetterGameState, pickle_game_state)
```

```
state = BetterGameState()
serialized = pickle.dumps(state)
print(serialized)
```

```
>>>
```

```
b'\x80\x04\x95W\x00\x00\x00\x00\x00\x00\x00\x8c\x08__main__
➡\x94\x8c\x13unpickle_game_state\x94\x93\x94}\x94(\x8c
➡\x05level\x94K\x00\x8c\x06points\x94K\x00\x8c\x05magic\x94K
➡\x05\x8c\x07version\x94K\x02u\x85\x94R\x94.'
```

```
print(Decimal('1.45'))  
print(Decimal(1.45))
```

```
>>>
```

```
1.45
```

```
1.4499999999999999555910790149937383830547332763671875
```

```
rate = Decimal('0.05')
seconds = Decimal('5')
small_cost = rate * seconds / Decimal(60)
print(small_cost)
```

&gt;&gt;&gt;

0.00416666666666666666666666666667

```
from decimal import ROUND_UP
```

```
rounded = cost.quantize(Decimal('0.01'), rounding=ROUND_UP)
```

```
print(f'Rounded {cost} to {rounded}')
```

```
>>>
```

```
Rounded 5.365 to 5.37
```

```
rounded = small_cost.quantize(Decimal('0.01'),  
                               rounding=ROUND_UP)  
print(f'Rounded {small_cost} to {rounded}')  
  
>>>  
Rounded 0.00416666666666666666666666666667 to 0.01
```

```
from random import randint
```

```
max_size = 10**4
```

```
data = [randint(0, max_size) for _ in range(max_size)]
```

```
test = lambda: insertion_sort(data)
```

```
>>>
```

```
20003 function calls in 1.320 seconds
```

```
Ordered by: cumulative time
```

ncalls	totttime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	1.320	1.320	main.py:35(<lambda>)
1	0.003	0.003	1.320	1.320	main.py:10(insertion_sort)
10000	1.306	0.000	1.317	0.000	main.py:20(insert_value)
9992	0.011	0.000	0.011	0.000	{method 'insert' of 'list' objects}
8	0.000	0.000	0.000	0.000	{method 'append' of 'list' objects}



```
>>>
```

```
30003 function calls in 0.017 seconds
```

```
Ordered by: cumulative time
```

ncalls	totttime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	0.017	0.017	main.py:35(<lambda>)
1	0.002	0.002	0.017	0.017	main.py:10(insertion_sort)
10000	0.003	0.000	0.015	0.000	main.py:110(insert_value)
10000	0.008	0.000	0.008	0.000	{method 'insert' of 'list' objects}
10000	0.004	0.000	0.004	0.000	{built-in method _bisect.bisect_left}

```
>>>
```

```
20242 function calls in 0.118 seconds
```

```
Ordered by: cumulative time
```

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	0.118	0.118	main.py:176(my_program)
20	0.003	0.000	0.117	0.006	main.py:168(first_func)
20200	0.115	0.000	0.115	0.000	main.py:161(my_utility)
20	0.000	0.000	0.001	0.000	main.py:172(second_func)

```
>>>
```

```
Ordered by: cumulative time
```

Function	was called by...	ncalls	tottime	cumtime	
main.py:176(my_program)	<-				
main.py:168(first_func)	<-	20	0.003	0.117	main.py:176(my_program)
main.py:161(my_utility)	<-	20000	0.114	0.114	main.py:168(first_func)
		200	0.001	0.001	main.py:172(second_func)
Profiling.md:172(second_func)	<-	20	0.000	0.001	main.py:176(my_program)

```
class Email:
    def __init__(self, sender, receiver, message):
        self.sender = sender
        self.receiver = receiver
        self.message = message
    ...
```

```
class NoEmailError(Exception):  
    pass  
  
def try_receive_email():  
    # Returns an Email instance or raises NoEmailError  
    ...
```

```
def consume_one_email(queue):  
    if not queue:  
        return  
    email = queue.pop(0) # Consumer  
    # Index the message for long-term archival  
    ...
```

```
import timeit

def print_results(count, tests):
    avg_iteration = sum(tests) / len(tests)
    print(f'Count {count:>5,} takes {avg_iteration:.6f}s')
    return count, avg_iteration

def list_append_benchmark(count):
    def run(queue):
```

```
def print_delta(before, after):
    before_count, before_time = before
    after_count, after_time = after
    growth = 1 + (after_count - before_count) / before_count
    slowdown = 1 + (after_time - before_time) / before_time
    print(f'{growth:>4.1f}x data size, {slowdown:>4.1f}x time')
```

```
baseline = list_append_benchmark(500)
for count in (1_000, 2_000, 3_000, 4_000, 5_000):
    comparison = list_append_benchmark(count)
    print_delta(baseline, comparison)
```

```
>>>
```

```
Count    500 takes 0.000039s
```

```
Count 1,000 takes 0.000073s
      2.0x data size,  1.9x time
```

```
Count 2,000 takes 0.000121s
      4.0x data size,  3.1x time
```

```
Count 3,000 takes 0.000172s
      6.0x data size,  4.5x time
```

```
Count 4,000 takes 0.000240s
      8.0x data size,  6.2x time
```

```
Count 5,000 takes 0.000304s
     10.0x data size,  7.9x time
```



```
baseline = list_pop_benchmark(500)
for count in (1_000, 2_000, 3_000, 4_000, 5_000):
    comparison = list_pop_benchmark(count)
    print_delta(baseline, comparison)
```

```
>>>
```

```
Count    500 takes 0.000050s
```

```
Count 1,000 takes 0.000133s
      2.0x data size,  2.7x time
```

```
Count 2,000 takes 0.000347s
      4.0x data size,  6.9x time
```

```
Count 3,000 takes 0.000663s
      6.0x data size, 13.2x time
```

```
Count 4,000 takes 0.000943s
      8.0x data size, 18.8x time
```

```
Count 5,000 takes 0.001481s
     10.0x data size, 29.5x time
```

```

def deque_append_benchmark(count):
    def prepare():
        return collections.deque()

    def run(queue):
        for i in range(count):
            queue.append(i)

    tests = timeit.repeat(
        setup='queue = prepare()',
        stmt='run(queue)',
        globals=globals(),
        repeat=1000,
        number=1)
    return print_results(count, tests)

baseline = deque_append_benchmark(500)
for count in (1_000, 2_000, 3_000, 4_000, 5_000):
    comparison = deque_append_benchmark(count)
    print_delta(baseline, comparison)

```

```
>>>
```

```
Count    500 takes 0.000029s
```

```
Count 1,000 takes 0.000059s
    2.0x data size,  2.1x time
```

```
Count 2,000 takes 0.000121s
    4.0x data size,  4.2x time
```

```
Count 3,000 takes 0.000171s
    6.0x data size,  6.0x time
```

```
Count 4,000 takes 0.000243s
    8.0x data size,  8.5x time
```

```
Count 5,000 takes 0.000295s
```

10.0x data size, 10.3x time

```

def dequeue_popleft_benchmark(count):
    def prepare():
        return collections.deque(range(count))

    def run(queue):
        while queue:
            queue.popleft()

    tests = timeit.repeat(
        setup='queue = prepare()',
        stmt='run(queue)',
        globals=locals(),
        repeat=1000,
        number=1)

    return print_results(count, tests)

baseline = dequeue_popleft_benchmark(500)
for count in (1_000, 2_000, 3_000, 4_000, 5_000):
    comparison = dequeue_popleft_benchmark(count)
    print_delta(baseline, comparison)

```

```
>>>
```

```
Count    500 takes 0.000024s
```

```
Count 1,000 takes 0.000050s
      2.0x data size,  2.1x time
```

```
Count 2,000 takes 0.000100s
      4.0x data size,  4.2x time
```

```
Count 3,000 takes 0.000152s
      6.0x data size,  6.3x time
```

```
Count 4,000 takes 0.000207s
      8.0x data size,  8.6x time
```

```
Count 5,000 takes 0.000265s
```

10.0x data size, 11.0x time

```
def find_closest(sequence, goal):  
    for index, value in enumerate(sequence):  
        if goal < value:  
            return index  
    raise ValueError(f'{goal} is out of bounds')  
  
index = find_closest(data, 91234.56)  
assert index == 91235
```

```
from bisect import bisect_left

index = bisect_left(data, 91234)      # Exact match
assert index == 91234

index = bisect_left(data, 91234.56)  # Closest match
assert index == 91235
```

```

import random
import timeit

size = 10**5
iterations = 1000

data = list(range(size))
to_lookup = [random.randint(0, size)
              for _ in range(iterations)]

def run_linear(data, to_lookup):
    for index in to_lookup:
        data.index(index)

def run_bisect(data, to_lookup):
    for index in to_lookup:
        bisect_left(data, index)

baseline = timeit.timeit(
    stmt='run_linear(data, to_lookup)',
    globals=globals(),
    number=10)
print(f'Linear search takes {baseline:.6f}s')

comparison = timeit.timeit(
    stmt='run_bisect(data, to_lookup)',
    globals=globals(),
    number=10)
print(f'Bisect search takes {comparison:.6f}s')

slowdown = 1 + ((baseline - comparison) / comparison)
print(f'{slowdown:.1f}x time')
>>>
Linear search takes 5.370117s
Bisect search takes 0.005220s

```



1028.7x time

```
def add_book(queue, book):  
    queue.append(book)  
    queue.sort(key=lambda x: x.due_date, reverse=True)
```

```
queue = []  
add_book(queue, Book('Don Quixote', '2019-06-07'))  
add_book(queue, Book('Frankenstein', '2019-06-05'))  
add_book(queue, Book('Les Misérables', '2019-06-08'))  
add_book(queue, Book('War and Peace', '2019-06-03'))
```

```
class NoOverdueBooks(Exception):  
    pass  
  
def next_overdue_book(queue, now):  
    if queue:  
        book = queue[-1]  
        if book.due_date < now:  
            queue.pop()  
            return book  
  
    raise NoOverdueBooks
```

```
def return_book(queue, book):  
    queue.remove(book)  
  
queue = []  
book = Book('Treasure Island', '2019-06-04')  
  
add_book(queue, book)  
print('Before return:', [x.title for x in queue])  
  
return_book(queue, book)  
print('After return: ', [x.title for x in queue])  
  
>>>  
Before return: ['Treasure Island']  
After return:  []
```

```
baseline = list_overdue_benchmark(500)
for count in (1_000, 1_500, 2_000):
    comparison = list_overdue_benchmark(count)
    print_delta(baseline, comparison)
```

```
>>>
```

```
Count    500 takes 0.001138s
```

```
Count 1,000 takes 0.003317s
      2.0x data size,  2.9x time
```

```
Count 1,500 takes 0.007744s
      3.0x data size,  6.8x time
```

```
Count 2,000 takes 0.014739s
      4.0x data size, 13.0x time
```

```
baseline = list_return_benchmark(500)
for count in (1_000, 1_500, 2_000):
    comparison = list_return_benchmark(count)
    print_delta(baseline, comparison)
```

```
>>>
```

```
Count    500 takes 0.000898s
```

```
Count 1,000 takes 0.003331s
      2.0x data size,  3.7x time
```

```
Count 1,500 takes 0.007674s
      3.0x data size,  8.5x time
```

```
Count 2,000 takes 0.013721s
      4.0x data size, 15.3x time
```

```
queue = []
add_book(queue, Book('Little Women', '2019-06-05'))
add_book(queue, Book('The Time Machine', '2019-05-30'))

>>>
Traceback ...
TypeError: '<' not supported between instances of 'Book' and
↳ 'Book'
```

```
queue = []  
add_book(queue, Book('Pride and Prejudice', '2019-06-01'))  
add_book(queue, Book('The Time Machine', '2019-05-30'))  
add_book(queue, Book('Crime and Punishment', '2019-06-06'))  
add_book(queue, Book('Wuthering Heights', '2019-06-12'))
```



```
queue = [  
    Book('Pride and Prejudice', '2019-06-01'),  
    Book('The Time Machine', '2019-05-30'),  
    Book('Crime and Punishment', '2019-06-06'),  
    Book('Wuthering Heights', '2019-06-12'),  
]  
queue.sort()
```

```
from heapq import heapify
```

```
queue = [  
    Book('Pride and Prejudice', '2019-06-01'),  
    Book('The Time Machine', '2019-05-30'),  
    Book('Crime and Punishment', '2019-06-06'),  
    Book('Wuthering Heights', '2019-06-12'),  
]  
heapify(queue)
```

```
from heapq import heappop

def next_overdue_book(queue, now):
    if queue:
        book = queue[0]                # Most overdue first
        if book.due_date < now:
            heappop(queue)             # Remove the overdue book
            return book

    raise NoOverdueBooks
```

```
now = '2019-06-02'
```

```
book = next_overdue_book(queue, now)  
print(book.title)
```

```
book = next_overdue_book(queue, now)  
print(book.title)
```

```
try:  
    next_overdue_book(queue, now)  
except NoOverdueBooks:  
    pass                # Expected  
else:  
    assert False        # Doesn't happen
```

```
>>>
```

```
The Time Machine
```

```
Pride and Prejudice
```

```
baseline = heap_overdue_benchmark(500)
for count in (1_000, 1_500, 2_000):
    comparison = heap_overdue_benchmark(count)
    print_delta(baseline, comparison)
```

```
>>>
```

```
Count    500 takes 0.000150s
```

```
Count 1,000 takes 0.000325s
      2.0x data size,  2.2x time
```

```
Count 1,500 takes 0.000528s
      3.0x data size,  3.5x time
```

```
Count 2,000 takes 0.000658s
      4.0x data size,  4.4x time
```

```
def timecode_to_index(video_id, timecode):  
    ...  
    # Returns the byte offset in the video data  
  
def request_chunk(video_id, byte_offset, size):  
    ...  
    # Returns size bytes of video_id's data from the offset  
  
video_id = ...  
timecode = '01:09:14:28'  
byte_offset = timecode_to_index(video_id, timecode)  
size = 20 * 1024 * 1024  
video_data = request_chunk(video_id, byte_offset, size)
```

```
socket = ...           # socket connection to client
video_data = ...       # bytes containing data for video_id
byte_offset = ...      # Requested starting position
size = 20 * 1024 * 1024 # Requested chunk size

chunk = video_data[byte_offset:byte_offset + size]
socket.send(chunk)
```

```
import timeit

def run_test():
    chunk = video_data[byte_offset:byte_offset + size]
    # Call socket.send(chunk), but ignoring for benchmark

result = timeit.timeit(
    stmt='run_test()',
    globals=globals(),
    number=100) / 100

print(f'{result:0.9f} seconds')

>>>
0.004925669 seconds
```



```
data = b'shave and a haircut, two bits'
view = memoryview(data)
chunk = view[12:19]
print(chunk)
print('Size:          ', chunk.nbytes)
print('Data in view:   ', chunk.tobytes())
print('Underlying data:', chunk.obj)

>>>
<memory at 0x10951fb80>
Size:          7
Data in view:   b'haircut'
Underlying data: b'shave and a haircut, two bits'
```

```
video_view = memoryview(video_data)

def run_test():
    chunk = video_view[byte_offset:byte_offset + size]
    # Call socket.send(chunk), but ignoring for benchmark
result = timeit.timeit(
    stmt='run_test()',
    globals=globals(),
    number=100) / 100

print(f'{result:0.9f} seconds')

>>>
0.000000250 seconds
```

```
socket = ...          # socket connection to the client
video_cache = ...     # Cache of incoming video stream
byte_offset = ...     # Incoming buffer position
size = 1024 * 1024    # Incoming chunk size

chunk = socket.recv(size)
video_view = memoryview(video_cache)
before = video_view[:byte_offset]
after = video_view[byte_offset + size:]
new_cache = b''.join([before, chunk, after])
```

```
def run_test():
    chunk = socket.recv(size)
    before = video_view[:byte_offset]
    after = video_view[byte_offset + size:]
    new_cache = b''.join([before, chunk, after])
result = timeit.timeit(
    stmt='run_test()',
    globals=globals(),
    number=100) / 100

print(f'{result:0.9f} seconds')

>>>
0.033520550 seconds
```

```
my_bytes = b'hello'  
my_bytes[0] = b'\x79'
```

```
>>>
```

```
Traceback ...
```

```
TypeError: 'bytes' object does not support item assignment
```

```
my_array = bytearray(b'row, row, row your boat')
my_view = memoryview(my_array)
write_view = my_view[3:13]
write_view[:] = b'-10 bytes-'
print(my_array)

>>>
bytearray(b'row-10 bytes- your boat')
```

```
video_array = bytearray(video_cache)
write_view = memoryview(video_array)
chunk = write_view[byte_offset:byte_offset + size]
socket.recv_into(chunk)
```

```
def run_test():
    chunk = write_view[byte_offset:byte_offset + size]
    socket.recv_into(chunk)

result = timeit.timeit(
    stmt='run_test()',
    globals=globals(),
    number=100) / 100

print(f'{result:0.9f} seconds')

>>>
0.000033925 seconds
```



```
print(5)
print('5')
```

```
int_value = 5
str_value = '5'
print(f'{int_value} == {str_value} ?')
```

```
>>>
```

```
5
```

```
5
```

```
5 == 5 ?
```

```
print('%r' % 5)  
print('%r' % '5')
```

```
int_value = 5  
str_value = '5'  
print(f'{int_value!r} != {str_value!r}')
```

```
>>>
```

```
5
```

```
'5'
```

```
5 != '5'
```

```
class OpaqueClass:
    def __init__(self, x, y):
        self.x = x
        self.y = y

obj = OpaqueClass(1, 'foo')
print(obj)

>>>
<__main__.OpaqueClass object at 0x10963d6d0>
```

```
class BetterClass:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __repr__(self):
        return f'BetterClass({self.x!r}, {self.y!r})'
```

```
# utils.py
def to_str(data):
    if isinstance(data, str):
        return data
    elif isinstance(data, bytes):
        return data.decode('utf-8')
    else:
        raise TypeError('Must supply str or bytes, '
                        'found: %r' % data)
```

```
# utils_test.py
from unittest import TestCase, main
from utils import to_str

class UtilsTestCase(TestCase):
    def test_to_str_bytes(self):
        self.assertEqual('hello', to_str(b'hello'))

    def test_to_str_str(self):
        self.assertEqual('hello', to_str('hello'))

    def test_failing(self):
        self.assertEqual('incorrect', to_str('hello'))

if __name__ == '__main__':
    main()
```

```
$ python3 utils_test.py
```

```
F..
```

```
=====
FAIL: test_failing (__main__.UtilsTestCase)
-----
```

```
Traceback (most recent call last):
```

```
  File "utils_test.py", line 15, in test_failing
```

```
    self.assertEqual('incorrect', to_str('hello'))
```

```
AssertionError: 'incorrect' != 'hello'
```

```
- incorrect
```

```
+ hello
```

```
-----
Ran 3 tests in 0.002s
```

```
FAILED (failures=1)
```

```
$ python3 utils_test.py UtilsTestCase.test_to_str_bytes
```

```
.
```

```
-----
```

```
Ran 1 test in 0.000s
```

```
OK
```



```
# assert_test.py
from unittest import TestCase, main
from utils import to_str

class AssertTestCase(TestCase):
    def test_assert_helper(self):
        expected = 12
        found = 2 * 5
        self.assertEqual(expected, found)

    def test_assert_statement(self):
        expected = 12
        found = 2 * 5
        assert expected == found

if __name__ == '__main__':
    main()
```

```
$ python3 assert_test.py
```

```
FF
```

```
=====
FAIL: test_assert_helper (__main__.AssertTestCase)
-----
```

```
Traceback (most recent call last):
```

```
  File "assert_test.py", line 16, in test_assert_helper
    self.assertEqual(expected, found)
```

```
AssertionError: 12 != 10
```

```
=====
FAIL: test_assert_statement (__main__.AssertTestCase)
-----
```

```
Traceback (most recent call last):
```

```
  File "assert_test.py", line 11, in test_assert_statement
    assert expected == found
```

```
AssertionError
```

```
-----
Ran 2 tests in 0.001s
```

```
FAILED (failures=2)
```

```
-  
# utils_error_test.py  
from unittest import TestCase, main  
from utils import to_str  
  
class UtilsErrorTestCase(TestCase):  
    def test_to_str_bad(self):  
        with self.assertRaises(TypeError):  
            to_str(object())  
  
    def test_to_str_bad_encoding(self):  
        with self.assertRaises(UnicodeDecodeError):  
            to_str(b'\xfa\xfa')  
  
if __name__ == '__main__':  
    main()
```

```

# helper_test.py
from unittest import TestCase, main

def sum_squares(values):
    cumulative = 0
    for value in values:
        cumulative += value ** 2
    yield cumulative

class HelperTestCase(TestCase):
    def verify_complex_case(self, values, expected):
        expect_it = iter(expected)
        found_it = iter(sum_squares(values))
        test_it = zip(expect_it, found_it)

        for i, (expect, found) in enumerate(test_it):
            self.assertEqual(
                expect,
                found,
                f'Index {i} is wrong')

        # Verify both generators are exhausted
        try:
            next(expect_it)
        except StopIteration:
            pass
        else:
            self.fail('Expected longer than found')

        try:
            next(found_it)
        except StopIteration:
            pass
        else:
            self.fail('Found longer than expected')

    def test_wrong_lengths(self):
        values = [1.1, 2.2, 3.3]
        expected = [
            1.1**2,
        ]
        self.verify_complex_case(values, expected)

    def test_wrong_results(self):
        values = [1.1, 2.2, 3.3]
        expected = [
            1.1**2,
            1.1**2 + 2.2**2,
            1.1**2 + 2.2**2 + 3.3**2 + 4.4**2,
        ]
        self.verify_complex_case(values, expected)

if __name__ == '__main__':

```

```
11  __name__ == '__main__':  
12      main()
```

```
$ python3 helper_test.py
```

```
FF
```

```
=====
FAIL: test_wrong_lengths (__main__.HelperTestCase)
-----
```

```
Traceback (most recent call last):
```

```
  File "helper_test.py", line 43, in test_wrong_lengths
```

```
    self.verify_complex_case(values, expected)
```

```
  File "helper_test.py", line 34, in verify_complex_case
```

```
    self.fail('Found longer than expected')
```

```
AssertionError: Found longer than expected
```

```
=====
FAIL: test_wrong_results (__main__.HelperTestCase)
-----
```

```
Traceback (most recent call last):
```

```
  File "helper_test.py", line 52, in test_wrong_results
```

```
    self.verify_complex_case(values, expected)
```

```
  File "helper_test.py", line 24, in verify_complex_case
```

```
    f'Index {i} is wrong')
```

```
AssertionError: 36.3 != 16.939999999999998 : Index 2 is wrong
```

```
-----
Ran 2 tests in 0.002s
```

```
FAILED (failures=2)
```

```

# data_driven_test.py
from unittest import TestCase, main
from utils import to_str

class DataDrivenTestCase(TestCase):
    def test_good(self):
        good_cases = [
            (b'my bytes', 'my bytes'),
            ('no error', b'no error'), # This one will fail
            ('other str', 'other str'),
            ...
        ]
        for value, expected in good_cases:
            with self.subTest(value):
                self.assertEqual(expected, to_str(value))
    def test_bad(self):
        bad_cases = [
            (object(), TypeError),
            (b'\xfa\xfa', UnicodeDecodeError),
            ...
        ]
        for value, exception in bad_cases:
            with self.subTest(value):
                with self.assertRaises(exception):
                    to_str(value)

if __name__ == '__main__':
    main()

```

```
$ python3 data_driven_test.py
```

```
.
```

```
=====
FAIL: test_good (__main__.DataDrivenTestCase) [no error]
-----
```

```
Traceback (most recent call last):
```

```
  File "testing/data_driven_test.py", line 18, in test_good
```

```
    self.assertEqual(expected, to_str(value))
```

```
AssertionError: b'no error' != 'no error'
```

```
-----
Ran 2 tests in 0.001s
```

```
FAILED (failures=1)
```



```
# environment_test.py
from pathlib import Path
from tempfile import TemporaryDirectory
from unittest import TestCase, main

class EnvironmentTest(TestCase):
    def setUp(self):
        self.test_dir = TemporaryDirectory()
        self.test_path = Path(self.test_dir.name)

    def tearDown(self):
        self.test_dir.cleanup()

    def test_modify_file(self):
        with open(self.test_path / 'data.bin', 'w') as f:
            ...

if __name__ == '__main__':
    main()
```

```

# integration_test.py
from unittest import TestCase, main

def setUpModule():
    print('* Module setup')

def tearDownModule():
    print('* Module clean-up')

class IntegrationTest(TestCase):
    def setUp(self):
        print('* Test setup')

    def tearDown(self):
        print('* Test clean-up')

    def test_end_to_end1(self):
        print('* Test 1')

    def test_end_to_end2(self):
        print('* Test 2')

if __name__ == '__main__':
    main()

```

```

$ python3 integration_test.py
* Module setup
* Test setup
* Test 1
* Test clean-up
.* Test setup
* Test 2
* Test clean-up
.* Module clean-up

```

---

Ran 2 tests in 0.000s

OK

```
class DatabaseConnection:
    ...

def get_animals(database, species):
    # Query the database
    ...
    # Return a list of (name, last_mealtime) tuples
```

```
database = DatabaseConnection('localhost', '4444')
```

```
get_animals(database, 'Meerkat')
```

```
>>>
```

```
Traceback ...
```

```
DatabaseConnectionError: Not connected
```

```
from datetime import datetime
from unittest.mock import Mock

mock = Mock(spec=get_animals)
expected = [
    ('Spot', datetime(2019, 6, 5, 11, 15)),
    ('Fluffy', datetime(2019, 6, 5, 12, 30)),
    ('Jojo', datetime(2019, 6, 5, 12, 45)),
]
mock.return_value = expected
```

```
mock.does_not_exist
```

```
>>>
```

```
Traceback ...
```

```
AttributeError: Mock object has no attribute 'does_not_exist'
```

```
mock.assert_called_once_with(database, 'Meerkat')
```

```
mock.assert_called_once_with(database, 'Giraffe')
```

```
>>>
```

```
Traceback ...
```

```
AssertionError: expected call not found.
```

```
Expected: mock(<object object at 0x109038790>, 'Giraffe')
```

```
Actual: mock(<object object at 0x109038790>, 'Meerkat')
```



```
from unittest.mock import ANY
mock = Mock(spec=get_animals)
mock('database 1', 'Rabbit')
mock('database 2', 'Bison')
mock('database 3', 'Meerkat')

mock.assert_called_with(ANY, 'Meerkat')
```

```
class MyError(Exception):  
    pass
```

```
mock = Mock(spec=get_animals)  
mock.side_effect = MyError('Whoops! Big problem')  
result = mock(database, 'Meerkat')
```

```
>>>
```

```
Traceback ...
```

```
MyError: Whoops! Big problem
```

```
def get_food_period(database, species):
    # Query the database
    ...
    # Return a time delta

def feed_animal(database, name, when):
    # Write to the database
    ...

def do_rounds(database, species):
    now = datetime.datetime.utcnow()
    feeding_timedelta = get_food_period(database, species)
    animals = get_animals(database, species)
    fed = 0
    for name, last_mealtime in animals:
        if (now - last_mealtime) > feeding_timedelta:
            feed_animal(database, name, now)
            fed += 1

    return fed
```

```
def do_rounds(database, species, *,
              now_func=datetime.utcnow,
              food_func=get_food_period,
              animals_func=get_animals,
              feed_func=feed_animal):
    now = now_func()
    feeding_timedelta = food_func(database, species)
    animals = animals_func(database, species)
    fed = 0

    for name, last_mealtime in animals:
        if (now - last_mealtime) > feeding_timedelta:
            feed_func(database, name, now)
            fed += 1

    return fed
```

```
from datetime import timedelta
now_func = Mock(spec=datetime.utcnow)
now_func.return_value = datetime(2019, 6, 5, 15, 45)

food_func = Mock(spec=get_food_period)
food_func.return_value = timedelta(hours=3)

animals_func = Mock(spec=get_animals)
animals_func.return_value = [
    ('Spot', datetime(2019, 6, 5, 11, 15)),
    ('Fluffy', datetime(2019, 6, 5, 12, 30)),
    ('Jojo', datetime(2019, 6, 5, 12, 45)),
]

feed_func = Mock(spec=feed_animal)
```

```
from unittest.mock import call

food_func.assert_called_once_with(database, 'Meerkat')

animals_func.assert_called_once_with(database, 'Meerkat')

feed_func.assert_has_calls(
    [
        call(database, 'Spot', now_func.return_value),
        call(database, 'Fluffy', now_func.return_value),
    ],
    any_order=True)
```

```
from unittest.mock import patch

print('Outside patch:', get_animals)

with patch('__main__.get_animals'):
    print('Inside patch: ', get_animals)

print('Outside again:', get_animals)

>>>
Outside patch: <function get_animals at 0x109217040>
Inside patch:  <MagicMock name='get_animals' id='4454622832'>
Outside again: <function get_animals at 0x109217040>
```

```
fake_now = datetime(2019, 6, 5, 15, 45)
```

```
with patch('datetime.datetime.utcnow'):  
    datetime.utcnow.return_value = fake_now
```

```
>>>
```

```
Traceback ...
```

```
TypeError: can't set attributes of built-in/extension type
```

```
↳ 'datetime.datetime'
```



```
def get_do_rounds_time():  
    return datetime.datetime.utcnow()  
  
def do_rounds(database, species):  
    now = get_do_rounds_time()  
    ...  
  
with patch('__main__.get_do_rounds_time'):  
    ...
```

```
def do_rounds(database, species, *, utcnow=datetime.utcnow):
    now = utcnow()
    feeding_timedelta = get_food_period(database, species)
    animals = get_animals(database, species)
    fed = 0

    for name, last_mealtime in animals:
        if (now - last_mealtime) > feeding_timedelta:
            feed_func(database, name, now)
            fed += 1

    return fed
```

```
from unittest.mock import DEFAULT

with patch.multiple('__main__',
                    autospec=True,
                    get_food_period=DEFAULT,
                    get_animals=DEFAULT,
                    feed_animal=DEFAULT):
    now_func = Mock(spec=datetime.utcnow)
    now_func.return_value = datetime(2019, 6, 5, 15, 45)
    get_food_period.return_value = timedelta(hours=3)
    get_animals.return_value = [
        ('Spot', datetime(2019, 6, 5, 11, 15)),
        ('Fluffy', datetime(2019, 6, 5, 12, 30)),
        ('Jojo', datetime(2019, 6, 5, 12, 45))
    ]
```

```
result = do_rounds(database, 'Meerkat', utcnow=now_func)
assert result == 2
```

```
food_func.assert_called_once_with(database, 'Meerkat')
animals_func.assert_called_once_with(database, 'Meerkat')
feed_func.assert_has_calls(
    [
        call(database, 'Spot', now_func.return_value),
        call(database, 'Fluffy', now_func.return_value),
    ],
    any_order=True)
```

```
class ZooDatabase:
    ...

    def get_animals(self, species):
        ...

    def get_food_period(self, species):
        ...

    def feed_animal(self, name, when):
        ...
```

```
from datetime import datetime
```

```
def do_rounds(database, species, *, utcnow=datetime.utcnow):  
    now = utcnow()  
    feeding_timedelta = database.get_food_period(species)  
    animals = database.get_animals(species)  
    fed = 0  
  
    for name, last_mealtime in animals:  
        if (now - last_mealtime) >= feeding_timedelta:  
            database.feed_animal(name, now)  
            fed += 1  
  
    return fed
```

```
from unittest.mock import Mock

database = Mock(spec=ZooDatabase)
print(database.feed_animal)
database.feed_animal()
database.feed_animal.assert_any_call()

>>>
<Mock name='mock.feed_animal' id='4384773408'>
```

```
from datetime import timedelta
from unittest.mock import call

now_func = Mock(spec=datetime.utcnow)
now_func.return_value = datetime(2019, 6, 5, 15, 45)

database = Mock(spec=ZooDatabase)
database.get_food_period.return_value = timedelta(hours=3)
database.get_animals.return_value = [
    ('Spot', datetime(2019, 6, 5, 11, 15)),
    ('Fluffy', datetime(2019, 6, 5, 12, 30)),
    ('Jojo', datetime(2019, 6, 5, 12, 55))
]
```



```
result = do_rounds(database, 'Meerkat', utcnow=now_func)
assert result == 2
```

```
database.get_food_period.assert_called_once_with('Meerkat')
database.get_animals.assert_called_once_with('Meerkat')
database.feed_animal.assert_has_calls(
    [
        call('Spot', now_func.return_value),
        call('Fluffy', now_func.return_value),
    ],
    any_order=True)
```

```
database.bad_method_name()
```

```
>>>
```

```
Traceback ...
```

```
AttributeError: Mock object has no attribute 'bad_method_name'
```

```
DATABASE = None
```

```
def get_database():  
    global DATABASE  
    if DATABASE is None:  
        DATABASE = ZooDatabase()  
    return DATABASE
```

```
def main(argv):  
    database = get_database()  
    species = argv[1]  
    count = do_rounds(database, species)  
    print(f'Fed {count} {species}(s)')  
    return 0
```

```

import contextlib
import io
from unittest.mock import patch
with patch('__main__.DATABASE', spec=ZooDatabase):
    now = datetime.utcnow()

    DATABASE.get_food_period.return_value = timedelta(hours=3)
    DATABASE.get_animals.return_value = [
        ('Spot', now - timedelta(minutes=4.5)),
        ('Fluffy', now - timedelta(hours=3.25)),
        ('Jojo', now - timedelta(hours=3)),
    ]

    fake_stdout = io.StringIO()
    with contextlib.redirect_stdout(fake_stdout):
        main(['program name', 'Meerkat'])

    found = fake_stdout.getvalue()
    expected = 'Fed 2 Meerkat(s)\n'

    assert found == expected

```

```
# always_breakpoint.py
import math

def compute_rmse(observed, ideal):
    total_err_2 = 0
    count = 0
    for got, wanted in zip(observed, ideal):
        err_2 = (got - wanted) ** 2
        breakpoint() # Start the debugger here
        total_err_2 += err_2
        count += 1

    mean_err = total_err_2 / count
    rmse = math.sqrt(mean_err)
    return rmse

result = compute_rmse(
    [1.8, 1.7, 3.2, 6],
    [2, 1.5, 3, 5])
print(result)
```

```
$ python3 always_breakpoint.py  
> always_breakpoint.py(12)compute_rmse()  
-> total_err_2 += err_2  
(Pdb)
```

```
# conditional_breakpoint.py
def compute_rmse(observed, ideal):
    ...
    for got, wanted in zip(observed, ideal):
        err_2 = (got - wanted) ** 2
        if err_2 >= 1: # Start the debugger if True
            breakpoint()
        total_err_2 += err_2
        count += 1
    ...
result = compute_rmse(
    [1.8, 1.7, 3.2, 7],
    [2, 1.5, 3, 5])
print(result)
```

```
$ python3 conditional_breakpoint.py  
> conditional_breakpoint.py(14)compute_rmse()  
-> total_err_2 += err_2  
(Pdb) wanted  
5  
(Pdb) got  
7  
(Pdb) err_2  
4
```



```
# postmortem_breakpoint.py
import math

def compute_rmse(observed, ideal):
    ...

result = compute_rmse(
    [1.8, 1.7, 3.2, 7j], # Bad input
    [2, 1.5, 3, 5])
print(result)
```

```
$ python3 -m pdb -c continue postmortem_breakpoint.py
Traceback (most recent call last):
  File ".../pdb.py", line 1697, in main
    pdb._runscript(mainpyfile)
  File ".../pdb.py", line 1566, in _runscript
    self.run(statement)
  File ".../bdb.py", line 585, in run
    exec(cmd, globals, locals)
  File "<string>", line 1, in <module>
  File "postmortem_breakpoint.py", line 4, in <module>
    import math
  File "postmortem_breakpoint.py", line 16, in compute_rmse
    rmse = math.sqrt(mean_err)
TypeError: can't convert complex to float
Uncaught exception. Entering post mortem debugging
Running 'cont' or 'step' will restart the program
> postmortem_breakpoint.py(16)compute_rmse()
-> rmse = math.sqrt(mean_err)
(Pdb) mean_err
(-5.97-17.5j)
```

```
$ python3
>>> import my_module
>>> my_module.compute_stddev([5])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "my_module.py", line 17, in compute_stddev
    variance = compute_variance(data)
  File "my_module.py", line 13, in compute_variance
    variance = err_2_sum / (len(data) - 1)
ZeroDivisionError: float division by zero
>>> import pdb; pdb.pm()
> my_module.py(13)compute_variance()
-> variance = err_2_sum / (len(data) - 1)
(Pdb) err_2_sum
0.0
(Pdb) len(data)
1
```

```
# waste_memory.py
```

```
import os
```

```
class MyObject:
```

```
    def __init__(self):
```

```
        self.data = os.urandom(100)
```

```
def get_data():
```

```
    values = []
```

```
    for _ in range(100):
```

```
        obj = MyObject()
```

```
        values.append(obj)
```

```
    return values
```

```
def run():
```

```
    deep_values = []
```

```
    for _ in range(100):
```

```
        deep_values.append(get_data())
```

```
    return deep_values
```

```
# using_gc.py
```

```
import gc
```

```
found_objects = gc.get_objects()
```

```
print('Before:', len(found_objects))
```

```
import waste_memory
```

```
hold_reference = waste_memory.run()
```

```
found_objects = gc.get_objects()
```

```
print('After: ', len(found_objects))
```

```
for obj in found_objects[:3]:
```

```
    print(repr(obj)[:100])
```

```
>>>
```

```
Before: 6207
```

```
After: 16801
```

```
<waste_memory.MyObject object at 0x10390aeb8>
```

```
<waste_memory.MyObject object at 0x10390aef0>
```

```
<waste_memory.MyObject object at 0x10390af28>
```

```
...
```

```

# top_n.py
import tracemalloc

tracemalloc.start(10)                                # Set stack depth
time1 = tracemalloc.take_snapshot()                   # Before snapshot

import waste_memory

x = waste_memory.run()                                # Usage to debug
time2 = tracemalloc.take_snapshot()                   # After snapshot

stats = time2.compare_to(time1, 'lineno')             # Compare snapshots
for stat in stats[:3]:
    print(stat)

>>>
waste_memory.py:5: size=2392 KiB (+2392 KiB), count=29994
➡(+29994), average=82 B
waste_memory.py:10: size=547 KiB (+547 KiB), count=10001
➡(+10001), average=56 B
waste_memory.py:11: size=82.8 KiB (+82.8 KiB), count=100
➡(+100), average=848 B

```

```
# with_trace.py
import tracemalloc

tracemalloc.start(10)
time1 = tracemalloc.take_snapshot()

import waste_memory

x = waste_memory.run()
time2 = tracemalloc.take_snapshot()

stats = time2.compare_to(time1, 'traceback')
top = stats[0]
print('Biggest offender is:')
print('\n'.join(top.traceback.format()))

>>>
Biggest offender is:
  File "with_trace.py", line 9
    x = waste_memory.run()
  File "waste_memory.py", line 17
    deep_values.append(get_data())
  File "waste_memory.py", line 10
    obj = MyObject()
  File "waste_memory.py", line 5
    self.data = os.urandom(100)
```

```
$ python3 -m pip show Sphinx
```

```
Name: Sphinx
```

```
Version: 2.1.2
```

```
Summary: Python documentation generator
```

```
Location: /usr/local/lib/python3.8/site-packages
```

```
Requires: alabaster, imagesize, requests,
```

```
➡sphinxcontrib-applehelp, sphinxcontrib-qthelp,
```

```
➡Jinja2, setuptools, sphinxcontrib-jsmath,
```

```
➡sphinxcontrib-serializinghtml, Pygments, snowballstemmer,
```

```
➡packaging, sphinxcontrib-devhelp, sphinxcontrib-htmlhelp,
```

```
➡babel, docutils
```

```
Required-by:
```



```
$ python3 -m pip show flask
```

```
Name: Flask
```

```
Version: 1.0.3
```

```
Summary: A simple framework for building complex web applications.
```

```
Location: /usr/local/lib/python3.8/site-packages
```

```
Requires: itsdangerous, click, Jinja2, Werkzeug
```

```
Required-by:
```

```
$ python3 -m venv myproject
```

```
$ cd myproject
```

```
$ ls
```

```
bin      include  lib      pyvenv.cfg
```

```
PS C:\> myproject\Scripts\activate.ps1  
(myproject) PS C:>
```

```
(myproject)$ which python3  
/tmp/myproject/bin/python3  
(myproject)$ ls -l /tmp/myproject/bin/python3  
... -> /usr/local/bin/python3.8
```

```
(myproject)$ python3 -c 'import pytz'
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ModuleNotFoundError: No module named 'pytz'
```

```
(myproject)$ python3 -m pip install pytz
Collecting pytz
  Downloading ...
Installing collected packages: pytz
Successfully installed pytz-2019.1
```

```
(myproject)$ python3 -m pip freeze > requirements.txt
(myproject)$ cat requirements.txt
certifi==2019.3.9
chardet==3.0.4
idna==2.8
numpy==1.16.2
pytz==2018.9
requests==2.21.0
urllib3==1.24.1
```

```
(otherproject)$ python3 -m pip list
```

Package	Version
---------	---------

-----	-----
-------	-------

pip	10.0.1
-----	--------

setuptools	39.0.1
------------	--------



```
(otherproject)$ python3 -m pip install -r /tmp/myproject/  
↳requirements.txt
```

```
(otherproject)$ python3 -m pip list
```

Package	Version
---------	---------

-----	-----
-------	-------

certifi	2019.3.9
---------	----------

chardet	3.0.4
---------	-------

idna	2.8
------	-----

numpy	1.16.2
-------	--------

pip	10.0.1
-----	--------

pytz	2018.9
------	--------

requests	2.21.0
----------	--------

setuptools	39.0.1
------------	--------

urllib3	1.24.1
---------	--------

```
def palindrome(word):  
    """Return True if the given word is a palindrome."""  
    return word == word[::-1]  
  
assert palindrome('tacocat')  
assert not palindrome('banana')
```

```
print(repr(palindrome.__doc__))
```

```
>>>
```

```
'Return True if the given word is a palindrome.'
```

```
$ python3 -m pydoc -p 1234
Server ready at http://localhost:1234/
Server commands: [b]rowser, [q]uit
server> b
```

```
# words.py
#!/usr/bin/env python3
"""Library for finding linguistic patterns in words.
```

Testing how words relate to each other can be tricky sometimes!  
This module provides easy ways to determine when words you've  
found have special properties.

Available functions:

- `palindrome`: Determine if a word is a palindrome.
- `check_anagram`: Determine if two words are anagrams.

```
...
"""
```

```
...
```

```
class Player:
    """Represents a player of the game.

    Subclasses may override the 'tick' method to provide
    custom animations for the player's movement depending
    on their power level, etc.

    Public attributes:
    - power: Unused power-ups (float between 0 and 1).
    - coins: Coins found during the level (integer).
    """
    ...
```

```
def find_anagrams(word, dictionary):  
    """Find all anagrams for a word.  
  
    This function only runs as fast as the test for  
    membership in the 'dictionary' container.  
  
    Args:  
        word: String of the target word.  
        dictionary: collections.abc.Container with all  
                   strings that are known to be actual words.  
  
    Returns:  
        List of anagrams that were found. Empty if  
        none were found.  
    """  
    ...
```



```
from typing import Container, List
```

```
def find_anagrams(word: str,  
                  dictionary: Container[str]) -> List[str]:  
    ...
```

```
def find_anagrams(word: str,  
                  dictionary: Container[str]) -> List[str]:
```

```
    """Find all anagrams for a word.
```

```
    This function only runs as fast as the test for  
    membership in the 'dictionary' container.
```

```
    Args:
```

```
        word: Target word.
```

```
        dictionary: All known actual words.
```

```
    Returns:
```

```
        Anagrams that were found.
```

```
    """
```

```
    ...
```

```
# main2.py
from analysis.utils import inspect
from frontend.utils import inspect # Overwrites!
```

```
# main2.py
from analysis.utils import inspect
from frontend.utils import inspect # Overwrites!
```

```
# main3.py
from analysis.utils import inspect as analysis_inspect
from frontend.utils import inspect as frontend_inspect

value = 33
if analysis_inspect(value) == frontend_inspect(value):
    print('Inspection equal!')
```

```
# main4.py
import analysis.utils
import frontend.utils

value = 33
if (analysis.utils.inspect(value) ==
    frontend.utils.inspect(value)):
    print('Inspection equal!')
```

```
# models.py
```

```
__all__ = ['Projectile']
```

```
class Projectile:
```

```
    def __init__(self, mass, velocity):
```

```
        self.mass = mass
```

```
        self.velocity = velocity
```

```
# db_connection.py
import sys

class Win32Database:
    ...

class PosixDatabase:
    ...

if sys.platform.startswith('win32'):
    Database = Win32Database
else:
    Database = PosixDatabase
```



```
# my_module.py
def determine_weight(volume, density):
    if density <= 0:
        raise ValueError('Density must be positive')
    ...
```

```
# my_module.py
class Error(Exception):
    """Base-class for all exceptions raised by this module."""

class InvalidDensityError(Error):
    """There was a problem with a provided density value."""

class InvalidVolumeError(Error):
    """There was a problem with the provided weight value."""

def determine_weight(volume, density):
    if density < 0:
        raise InvalidDensityError('Density must be positive')
    if volume < 0:
        raise InvalidVolumeError('Volume must be positive')
    if volume == 0:
        density / volume
```

```
try:
    weight = my_module.determine_weight(1, -1)
except my_module.Error:
    logging.exception('Unexpected error')

>>>
Unexpected error
Traceback (most recent call last):
  File ".../example.py", line 3, in <module>
    weight = my_module.determine_weight(1, -1)
  File ".../my_module.py", line 10, in determine_weight
    raise InvalidDensityError('Density must be positive')
InvalidDensityError: Density must be positive
```

```
try:
    weight = my_module.determine_weight(-1, 1)
except my_module.InvalidDensityError:
    weight = 0
except my_module.Error:
    logging.exception('Bug in the calling code')

>>>
Bug in the calling code
Traceback (most recent call last):
  File ".../example.py", line 3, in <module>
    weight = my_module.determine_weight(-1, 1)
  File ".../my_module.py", line 12, in determine_weight
    raise InvalidVolumeError('Volume must be positive')
InvalidVolumeError: Volume must be positive
```

```
try:
    weight = my_module.determine_weight(0, 1)
except my_module.InvalidDensityError:
    weight = 0
except my_module.Error:
    logging.exception('Bug in the calling code')
except Exception:
    logging.exception('Bug in the API code!')
    raise # Re-raise exception to the caller

>>>
Bug in the API code!
Traceback (most recent call last):
  File ".../example.py", line 3, in <module>
    weight = my_module.determine_weight(0, 1)
  File ".../my_module.py", line 14, in determine_weight
    density / volume
ZeroDivisionError: division by zero
Traceback ...
ZeroDivisionError: division by zero
```

```
# my_module.py
...

class NegativeDensityError(InvalidDensityError):
    """A provided density value was negative."""

...

def determine_weight(volume, density):
    if density < 0:
        raise NegativeDensityError('Density must be positive')
    ...
```

```
try:
    weight = my_module.determine_weight(1, -1)
except my_module.NegativeDensityError:
    raise ValueError('Must supply non-negative density')
except my_module.InvalidDensityError:
    weight = 0
except my_module.Error:
    logging.exception('Bug in the calling code')
except Exception:
    logging.exception('Bug in the API code!')
    raise
```

```
>>>
```

```
Traceback ...
```

```
NegativeDensityError: Density must be positive
```

The above exception was the direct cause of the following  
➡exception:

```
Traceback ...
```

```
ValueError: Must supply non-negative density
```

```
# my_module.py
class Error(Exception):
    """Base-class for all exceptions raised by this module."""

class WeightError(Error):
    """Base-class for weight calculation errors."""

class VolumeError(Error):
    """Base-class for volume calculation errors."""

class DensityError(Error):
    """Base-class for density calculation errors."""
...

```



```
# dialog.py
import app

class Dialog:
    def __init__(self, save_dir):
        self.save_dir = save_dir
        ...

save_dialog = Dialog(app.prefs.get('save_dir'))

def show():
    ...
```

```
# main.py
import app
```

I get an exception:

```
>>>
```

```
$ python3 main.py
```

```
Traceback (most recent call last):
```

```
  File ".../main.py", line 17, in <module>
```

```
    import app
```

```
  File ".../app.py", line 17, in <module>
```

```
    import dialog
```

```
  File ".../dialog.py", line 23, in <module>
```

```
    save_dialog = Dialog(app.prefs.get('save_dir'))
```

```
AttributeError: partially initialized module 'app' has no
➡attribute 'prefs' (most likely due to a circular import)
```

```
# dialog.py
import app

class Dialog:
    ...

save_dialog = Dialog()

def show():
    ...

def configure():
    save_dialog.save_dir = app.prefs.get('save_dir')
```

```
# dialog.py
class Dialog:
    ...

save_dialog = Dialog()

def show():
    import app # Dynamic import
    save_dialog.save_dir = app.prefs.get('save_dir')
    ...
```

```
def print_distance(speed, duration):  
    distance = speed * duration  
    print(f'{distance} miles')
```

```
print_distance(5, 2.5)
```

```
>>>
```

```
12.5 miles
```

```

CONVERSIONS = {
    'mph': 1.60934 / 3600 * 1000,    # m/s
    'hours': 3600,                  # seconds
    'miles': 1.60934 * 1000,        # m
    'meters': 1,                    # m
    'm/s': 1,                       # m
    'seconds': 1,                   # s
}

def convert(value, units):
    rate = CONVERSIONS[units]
    return rate * value

def localize(value, units):
    rate = CONVERSIONS[units]
    return value / rate

def print_distance(speed, duration, *,
                  speed_units='mph',
                  time_units='hours',
                  distance_units='miles'):
    norm_speed = convert(speed, speed_units)
    norm_duration = convert(duration, time_units)
    norm_distance = norm_speed * norm_duration
    distance = localize(norm_distance, distance_units)
    print(f'{distance} {distance_units}')

```

```
print_distance(1000, 3,  
               speed_units='meters',  
               time_units='seconds')
```

```
>>>
```

```
1.8641182099494205 miles
```

```
import warnings

def print_distance(speed, duration, *,
                  speed_units=None,
                  time_units=None,
                  distance_units=None):
    if speed_units is None:
        warnings.warn(
            'speed_units required', DeprecationWarning)
        speed_units = 'mph'

    if time_units is None:
        warnings.warn(
            'time_units required', DeprecationWarning)
        time_units = 'hours'

    if distance_units is None:
        warnings.warn(
            'distance_units required', DeprecationWarning)
        distance_units = 'miles'

    norm_speed = convert(speed, speed_units)
    norm_duration = convert(duration, time_units)
    norm_distance = norm_speed * norm_duration
    distance = localize(norm_distance, distance_units)
    print(f'{distance} {distance_units}')
```



```
import contextlib
import io

fake_stderr = io.StringIO()
with contextlib.redirect_stderr(fake_stderr):
    print_distance(1000, 3,
                  speed_units='meters',
                  time_units='seconds')

print(fake_stderr.getvalue())

>>>
1.8641182099494205 miles
.../example.py:97: DeprecationWarning: distance_units required
  warnings.warn(
```

```

def require(name, value, default):
    if value is not None:
        return value
    warnings.warn(
        f'{name} will be required soon, update your code',
        DeprecationWarning,
        stacklevel=3)
    return default

def print_distance(speed, duration, *,
                  speed_units=None,
                  time_units=None,
                  distance_units=None):
    speed_units = require('speed_units', speed_units, 'mph')
    time_units = require('time_units', time_units, 'hours')
    distance_units = require(
        'distance_units', distance_units, 'miles')

    norm_speed = convert(speed, speed_units)
    norm_duration = convert(duration, time_units)
    norm_distance = norm_speed * norm_duration
    distance = localize(norm_distance, distance_units)
    print(f'{distance} {distance_units}')

```

```
import contextlib
import io
```

```
fake_stderr = io.StringIO()
with contextlib.redirect_stderr(fake_stderr):
    print_distance(1000, 3,
                  speed_units='meters',
                  time_units='seconds')
```

```
print(fake_stderr.getvalue())
```

```
>>>
```

```
1.8641182099494205 miles
```

```
.../example.py:174: DeprecationWarning: distance_units will be
```

```
➡required soon, update your code
```

```
    print_distance(1000, 3,
```

```
warnings.simplefilter('error')
try:
    warnings.warn('This usage is deprecated',
                  DeprecationWarning)
except DeprecationWarning:
    pass # Expected
```

```
$ python -W error example_test.py
Traceback (most recent call last):
  File ".../example_test.py", line 6, in <module>
    warnings.warn('This might raise an exception!')
UserWarning: This might raise an exception!
```

```
warnings.simplefilter('ignore')  
warnings.warn('This will not be printed to stderr')
```

```
import logging

fake_stderr = io.StringIO()
handler = logging.StreamHandler(fake_stderr)
formatter = logging.Formatter(
    '%(asctime)-15s WARNING] %(message)s')
handler.setFormatter(formatter)

logging.captureWarnings(True)
logger = logging.getLogger('py.warnings')
logger.addHandler(handler)
logger.setLevel(logging.DEBUG)

warnings.resetwarnings()
warnings.simplefilter('default')
warnings.warn('This will go to the logs output')

print(fake_stderr.getvalue())

>>>
2019-06-11 19:48:19,132 WARNING] .../example.py:227:
➡UserWarning: This will go to the logs output
    warnings.warn('This will go to the logs output')
```

```
with warnings.catch_warnings(record=True) as found_warnings:  
    found = require('my_arg', None, 'fake units')  
    expected = 'fake units'  
    assert found == expected
```



```
assert len(found_warnings) == 1
single_warning = found_warnings[0]
assert str(single_warning.message) == (
    'my_arg will be required soon, update your code')
assert single_warning.category == DeprecationWarning
```

```
def subtract(a, b):  
    return a - b
```

```
subtract(10, '5')
```

```
>>>
```

```
Traceback ...
```

```
TypeError: unsupported operand type(s) for -: 'int' and 'str'
```

```
def subtract(a: int, b: int) -> int: # Function annotation
    return a - b
```

```
subtract(10, '5') # Oops: passed string value
```

```
$ python3 -m mypy --strict example.py
.../example.py:4: error: Argument 2 to "subtract" has
incompatible type "str"; expected "int"
```

```
def concat(a, b):  
    return a + b
```

```
concat('first', b'second')
```

```
>>>
```

```
Traceback ...
```

```
TypeError: can only concatenate str (not "bytes") to str
```

```
def concat(a: str, b: str) -> str:  
    return a + b
```

```
concat('first', b'second') # Oops: passed bytes value
```

```
$ python3 -m mypy --strict example.py
```

```
.../example.py:4: error: Argument 2 to "concat" has  
↳ incompatible type "bytes"; expected "str"
```

```
counter = Counter()  
counter.add(5)
```

```
>>>
```

```
Traceback ...
```

```
UnboundLocalError: local variable 'value' referenced before  
➡assignment
```

```
class Counter:
    def __init__(self) -> None:
        self.value: int = 0 # Field / variable annotation

    def add(self, offset: int) -> None:
        value += offset      # Oops: forgot "self."

    def get(self) -> int:
        self.value           # Oops: forgot "return"

counter = Counter()
counter.add(5)
counter.add(3)
assert counter.get() == 8

$ python3 -m mypy --strict example.py
.../example.py:6: error: Name 'value' is not defined
.../example.py:8: error: Missing return statement
```

```
from typing import Callable, List, TypeVar
```

```
Value = TypeVar('Value')
```

```
Func = Callable[[Value, Value], Value]
```

```
def combine(func: Func[Value], values: List[Value]) -> Value:  
    assert len(values) > 0
```

```
    result = values[0]  
    for next_value in values[1:]:  
        result = func(result, next_value)
```

```
    return result
```

```
Real = TypeVar('Real', int, float)
```

```
def add(x: Real, y: Real) -> Real:  
    return x + y
```

```
inputs = [1, 2, 3, 4j] # Oops: included a complex number  
result = combine(add, inputs)  
assert result == 10
```

```
$ python3 -m mypy --strict example.py
```

```
.../example.py:21: error: Argument 1 to "combine" has
```

```
↳ incompatible type "Callable[[Real, Real], Real]"; expected
```

```
↳ "Callable[[complex, complex], complex]"
```



```
from typing import Optional
```

```
def get_or_default(value: Optional[int],  
                  default: int) -> int:  
    if value is not None:  
        return value  
    return value # Oops: should have returned "default"
```

```
$ python3 -m mypy --strict example.py
```

```
.../example.py:7: error: Incompatible return value type (got  
↳ "None", expected "int")
```

```
class FirstClass:
    def __init__(self, value: SecondClass) -> None:
        self.value = value
```

```
class SecondClass:
    def __init__(self, value: int) -> None:
        self.value = value
```

```
second = SecondClass(5)
first = FirstClass(second)
```

```
$ python3 -m mypy --strict example.py
```

```
class FirstClass:
    def __init__(self, value: SecondClass) -> None: # Breaks
        self.value = value
```

```
class SecondClass:
    def __init__(self, value: int) -> None:
        self.value = value
```

```
second = SecondClass(5)
first = FirstClass(second)
```

```
>>>
```

```
Traceback ...
```

```
NameError: name 'SecondClass' is not defined
```

```
class FirstClass:
    def __init__(self, value: 'SecondClass') -> None: # OK
        self.value = value

class SecondClass:
    def __init__(self, value: int) -> None:
        self.value = value

second = SecondClass(5)
first = FirstClass(second)
```

```
from __future__ import annotations
```

```
class FirstClass:
```

```
    def __init__(self, value: SecondClass) -> None: # OK
        self.value = value
```

```
class SecondClass:
```

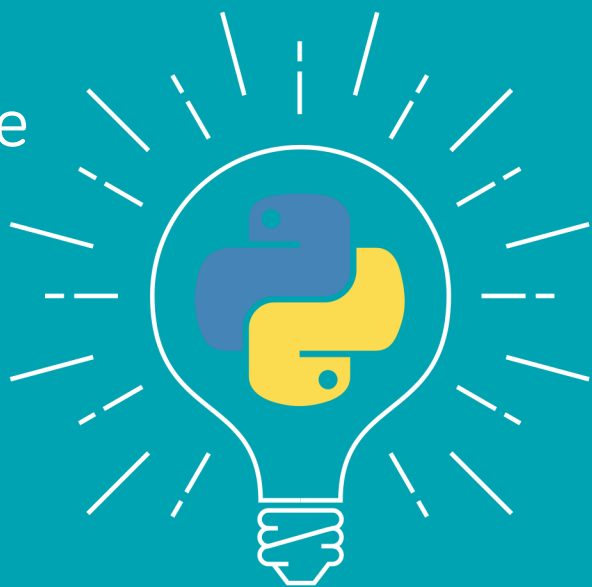
```
    def __init__(self, value: int) -> None:
        self.value = value
```

```
second = SecondClass(5)
```

```
first = FirstClass(second)
```

# PYTHON TRICKS THE BOOK

A Buffet  
of Awesome  
Python  
Features



Dan Bader

# Python Tricks: The Book

Dan Bader

For online information and ordering of this and other books by Dan Bader, please visit [dbader.org](http://dbader.org). For more information, please contact Dan Bader at [mail@dbader.org](mailto:mail@dbader.org).

Copyright © Dan Bader ([dbader.org](http://dbader.org)), 2016–2017

ISBN: 9781775093305 (paperback)

ISBN: 9781775093312 (electronic)

Cover design by Anja Pircher Design ([anjapircher.com](http://anjapircher.com))

“Python” and the Python logos are trademarks or registered trademarks of the Python Software Foundation, used by Dan Bader with permission from the Foundation.

Thank you for downloading this ebook. This ebook is licensed for your personal enjoyment only. This ebook may not be re-sold or given away to other people. If you would like to share this book with another person, please purchase an additional copy for each recipient. If you’re reading this book and did not purchase it, or it was not purchased for your use only, then please return to [dbader.org/pytricks-book](http://dbader.org/pytricks-book) and purchase your own copy. Thank you for respecting the hard work behind this book.

Updated 2017-10-27 I would like to thank Michael Howitz, Johnathan Willitts, Julian Orbach, Johnny Giorgis, Bob White, Daniel Meyer, Michael Stueben, Smital Desai, Andreas Kreisig, David Perkins, Jay Prakash Singh, and Ben Felder for their excellent feedback.



## What Pythonistas Say About *Python Tricks: The Book*

---

*"I love love love the book. It's like having a seasoned tutor explaining, well, tricks! I'm learning Python on the job and I'm coming from powershell, which I learned on the job—so lots of new, great stuff. Whenever I get stuck in Python (usually with flask blueprints or I feel like my code could be more Pythonic) I post questions in our internal Python chat room.*

*I'm often amazed at some of the answers coworkers give me. Dict comprehensions, lambdas, and generators often pepper their feedback. I am always impressed and yet flabbergasted at how powerful Python is when you know these tricks and can implement them correctly.*

*Your book was exactly what I wanted to help get me from a bewildered powershell scripter to someone who knows how and when to use these Pythonic 'tricks' everyone has been talking about.*

*As someone who doesn't have my degree in CS it's nice to have the text to explain things that others might have learned when they were classically educated. I am really enjoying the book and am subscribed to the emails as well, which is how I found out about the book."*

— **Daniel Meyer**, Sr. Desktop Administrator at Tesla Inc.

*"I first heard about your book from a co-worker who wanted to trick me with your example of how dictionaries are built. I was almost 100% sure about the reason why the end product was a much smaller/simpler dictionary but I must confess that I did not expect the outcome :)*

*He showed me the book via video conferencing and I sort of skimmed through it as he flipped the pages for me, and I was immediately curious to read more.*

*That same afternoon I purchased my own copy and proceeded to read your explanation for the way dictionaries are created in Python and later that day, as I met a different co-worker for coffee, I used the same trick on him :)*

*He then sprung a different question on the same principle, and because of the way you explained things in your book, I was able to not\* guess the result but correctly answer what the outcome would be. That means that you did a great job at explaining things :)\**

*I am not new in Python and some of the concepts in some of the chapters are not new to me, but I must say that I do get something out of every chapter so far, so kudos for writing a very nice book and for doing a fantastic job at explaining concepts behind the tricks! I'm very much looking forward to the updates and I will certainly let my friends and co-workers know about your book."*

— **Og Maciel**, Python Developer at Red Hat

*"I really enjoyed reading Dan's book. He explains important Python aspects with clear examples (using two twin cats to explain 'is' vs '==' for example).*

*It is not just code samples, it discusses relevant implementation details comprehensibly. What really matters though is that this book makes you write better Python code!*

*The book is actually responsible for recent new good Python habits I picked up, for example: using custom exceptions and ABC's (I found Dan's blog searching for abstract classes.) These new learnings alone are worth the price."*

— **Bob Belderbos**, Engineer at Oracle & Co-Founder of PyBites

# Contents

<b>Contents</b>	<b>6</b>
<b>Foreword</b>	<b>9</b>
<b>1 Introduction</b>	<b>11</b>
1.1 What’s a Python Trick? . . . . .	11
1.2 What This Book Will Do for You . . . . .	13
1.3 How to Read This Book . . . . .	14
<b>2 Patterns for Cleaner Python</b>	<b>15</b>
2.1 Covering Your A** With Assertions . . . . .	16
2.2 Complacent Comma Placement . . . . .	25
2.3 Context Managers and the with Statement . . . . .	29
2.4 Underscores, Dunders, and More . . . . .	36
2.5 A Shocking Truth About String Formatting . . . . .	48
2.6 “The Zen of Python” Easter Egg . . . . .	56
<b>3 Effective Functions</b>	<b>57</b>
3.1 Python’s Functions Are First-Class . . . . .	58
3.2 Lambdas Are Single-Expression Functions . . . . .	68
3.3 The Power of Decorators . . . . .	73
3.4 Fun With *args and **kwargs . . . . .	86
3.5 Function Argument Unpacking . . . . .	91
3.6 Nothing to Return Here . . . . .	94

<b>4</b>	<b>Classes &amp; OOP</b>	<b>97</b>
4.1	Object Comparisons: “is” vs “==” . . . . .	98
4.2	String Conversion (Every Class Needs a <code>__repr__</code> ) . . . . .	101
4.3	Defining Your Own Exception Classes . . . . .	111
4.4	Cloning Objects for Fun and Profit . . . . .	116
4.5	Abstract Base Classes Keep Inheritance in Check . . . . .	124
4.6	What Namedtuples Are Good For . . . . .	128
4.7	Class vs Instance Variable Pitfalls . . . . .	136
4.8	Instance, Class, and Static Methods Demystified . . . . .	143
<b>5</b>	<b>Common Data Structures in Python</b>	<b>153</b>
5.1	Dictionaries, Maps, and Hashtables . . . . .	156
5.2	Array Data Structures . . . . .	163
5.3	Records, Structs, and Data Transfer Objects . . . . .	173
5.4	Sets and Multisets . . . . .	185
5.5	Stacks (LIFOs) . . . . .	189
5.6	Queues (FIFOs) . . . . .	195
5.7	Priority Queues . . . . .	201
<b>6</b>	<b>Looping &amp; Iteration</b>	<b>205</b>
6.1	Writing Pythonic Loops . . . . .	206
6.2	Comprehending Comprehensions . . . . .	210
6.3	List Slicing Tricks and the Sushi Operator . . . . .	214
6.4	Beautiful Iterators . . . . .	218
6.5	Generators Are Simplified Iterators . . . . .	231
6.6	Generator Expressions . . . . .	239
6.7	Iterator Chains . . . . .	246
<b>7</b>	<b>Dictionary Tricks</b>	<b>250</b>
7.1	Dictionary Default Values . . . . .	251
7.2	Sorting Dictionaries for Fun and Profit . . . . .	255
7.3	Emulating Switch/Case Statements With Dicts . . . . .	259
7.4	The Craziest Dict Expression in the West . . . . .	264
7.5	So Many Ways to Merge Dictionaries . . . . .	271
7.6	Dictionary Pretty-Printing . . . . .	274

<b>8</b>	<b>Pythonic Productivity Techniques</b>	<b>277</b>
8.1	Exploring Python Modules and Objects . . . . .	278
8.2	Isolating Project Dependencies With Virtualenv . . .	282
8.3	Peeking Behind the Bytecode Curtain . . . . .	288
<b>9</b>	<b>Closing Thoughts</b>	<b>293</b>
9.1	Free Weekly Tips for Python Developers . . . . .	295
9.2	PythonistaCafe: A Community for Python Developers	296

# Foreword

It's been almost ten years since I first got acquainted with Python as a programming language. When I first learned Python many years ago, it was with a little reluctance. I had been programming in a different language before, and all of the sudden at work, I was assigned to a different team where everyone used Python. That was the beginning of my own Python journey.

When I was first introduced to Python, I was told that it was going to be easy, that I should be able to pick it up quickly. When I asked my colleagues for resources for learning Python, all they gave me was a link to Python's official documentation. Reading the documentation was confusing at first, and it really took me a while before I even felt comfortable navigating through it. Often I found myself needing to look for answers in StackOverflow.

Coming from a different programming language, I wasn't looking for just any resource for learning how to program or what classes and objects are. I was looking for specific resources that would teach me the features of Python, what sets it apart, and how writing in Python is different than writing code in another language.

It really has taken me many years to fully appreciate this language. As I read Dan's book, I kept thinking that I wished I had access to a book like this when I started learning Python many years ago.

For example, one of the many unique Python features that surprised me at first were list comprehensions. As Dan mentions in the book,

a tell of someone who just came to Python from a different language is the way they use for-loops. I recall one of the earliest code review comments I got when I started programming in Python was, “Why not use list comprehension here?” Dan explains this concept clearly in section 6, starting by showing how to loop the Pythonic way and building it all the way up to iterators and generators.

In chapter 2.5, Dan discusses the different ways to do string formatting in Python. String formatting is one of those things that defy the Zen of Python, that there should only be one obvious way to do things. Dan shows us the different ways, including my favorite new addition to the language, the f-strings, and he also explains the pros and cons of each method.

The Pythonic Productivity Techniques section is another great resource. It covers aspects beyond the Python programming language, and also includes tips on how to debug your programs, how to manage the dependencies, and gives you a peek inside Python bytecode.

It truly is an honor and my pleasure to introduce this book, Python Tricks, by my friend, Dan Bader.

By contributing to Python as a CPython core developer, I get connected to many members of the community. In my journey, I found mentors, allies, and made many new friends. They remind me that Python is not just about the code, Python is a community.

Mastering Python programming isn’t just about grasping the theoretical aspects of the language. It’s just as much about understanding and adopting the conventions and best practices used by its community.

Dan’s book will help you on this journey. I’m convinced that you’ll be more confident when writing Python programs after reading it.

— **Mariatta Wijaya**, Python Core Developer ([mariatta.ca](https://mariatta.ca))



# Chapter 1

## Introduction

### 1.1 What's a Python Trick?

**Python Trick:** *A short Python code snippet meant as a teaching tool. A Python Trick either teaches an aspect of Python with a simple illustration, or it serves as a motivating example, enabling you to dig deeper and develop an intuitive understanding.*

Python Tricks started out as a short series of code screenshots that I shared on Twitter for a week. To my surprise, they got rave responses and were shared and retweeted for days on end.

More and more developers started asking me for a way to “get the whole series.” Actually, I only had a few of these tricks lined up, spanning a variety of Python-related topics. There wasn’t a master plan behind them. They were just a fun little Twitter experiment.

But from these inquiries I got the sense that my short-and-sweet code examples would be worth exploring as a teaching tool. Eventually I set out to create a few more Python Tricks and shared them in an email series. Within a few days, several hundred Python developers had signed up and I was just blown away by that response.

Over the following days and weeks, a steady stream of Python developers reached out to me. They thanked me for making a part of the language they were struggling to understand *click* for them. Hearing this feedback felt awesome. I thought these Python Tricks were just code screenshots, but so many developers were getting a lot of value out of them.

That's when I decided to double down on my Python Tricks experiment and expanded it into a series of around 30 emails. Each of these was still just a headline and a code screenshot, and I soon realized the limits of that format. Around this time, a blind Python developer emailed me, disappointed to find that these Python Tricks were delivered as images he couldn't read with his screen reader.

Clearly, I needed to invest more time into this project to make it more appealing and more accessible to a wider audience. So, I sat down to re-create the whole series of Python Tricks emails in plain text and with proper HTML-based syntax highlighting. That new iteration of Python Tricks chugged along nicely for a while. Based on the responses I got, developers seemed happy they could finally copy and paste the code samples in order to play around with them.

As more and more developers signed up for the email series, I started noticing a pattern in the replies and questions I received. Some Tricks worked well as motivational examples by themselves. However, for the more complex ones there was no narrator to guide readers or to give them additional resources to develop a deeper understanding.

Let's just say this was another big area of improvement. My mission statement for [dbader.org](http://dbader.org) is to *help Python developers become more awesome*—and this was clearly an opportunity to get closer to that goal.

I decided to take the best and most valuable Python Tricks from the email course, and I started writing a new kind of Python book around them:

- A book that teaches the coolest aspects of the language with short and easy-to-digest examples.
- A book that works like a buffet of awesome Python features (yum!) and keeps motivation levels high.
- A book that takes you by the hand to guide you and help you deepen your understanding of Python.

This book is really a labor of love for me and also a huge experiment. I hope you'll enjoy reading it and learn something about Python in the process!

— Dan Bader

## 1.2 What This Book Will Do for You

My goal for this book is to make you a better—more effective, more knowledgeable, more practical—Python developer. You might be wondering, *How will reading this book help me achieve all that?*

*Python Tricks* is not a step-by-step Python tutorial. It is not an entry-level Python course. If you're in the beginning stages of learning Python, the book alone won't transform you into a professional Python developer. Reading it will still be beneficial to you, but you need to make sure you're working with some other resources to build up your foundational Python skills.

You'll get the most out of this book if you already have some knowledge of Python, and you want to get to the next level. It will work great for you if you've been coding Python for a while and you're ready to go deeper, to round out your knowledge, and to make your code more Pythonic.

Reading *Python Tricks* will also be great for you if you already have experience with other programming languages and you're looking to get up to speed with Python. You'll discover a ton of practical tips and design patterns that'll make you a more effective and skilled Python coder.

## 1.3 How to Read This Book

The best way to read *Python Tricks: The Book* is to treat it like a buffet of awesome Python features. Each Python Trick in the book is self-contained, so it's completely okay to jump straight to the ones that look the most interesting. In fact, I would encourage you to do just that.

Of course, you can also read through all the Python Tricks in the order they're laid out in the book. That way you won't miss any of them, and you'll know you've seen it all when you arrive at the final page.

Some of these tricks will be easy to understand right away, and you'll have no trouble incorporating them into your day to day work just by reading the chapter. Other tricks might require a bit more time to crack.

If you're having trouble making a particular trick work in your own programs, it helps to play through each of the code examples in a Python interpreter session.

If that doesn't make things click, then please feel free to reach out to me, so I can help you out and improve the explanation in this book. In the long run, that benefits not just you but all Pythonistas reading this book.

## **Chapter 2**

# **Patterns for Cleaner Python**

## 2.1 Covering Your A\*\* With Assertions

Sometimes a genuinely helpful language feature gets less attention than it deserves. For some reason, this is what happened to Python's built-in `assert` statement.

In this chapter I'm going to give you an introduction to using assertions in Python. You'll learn how to use them to help automatically detect errors in your Python programs. This will make your programs more reliable and easier to debug.

At this point, you might be wondering "What are assertions and what are they good for?" Let's get you some answers for that.

At its core, Python's `assert` statement is a debugging aid that tests a condition. If the `assert` condition is true, nothing happens, and your program continues to execute as normal. But if the condition evaluates to false, an `AssertionError` exception is raised with an optional error message.

### Assert in Python — An Example

Here's a simple example so you can see where assertions might come in handy. I tried to give this some semblance of a real-world problem you might actually encounter in one of your programs.

Suppose you were building an online store with Python. You're working to add a discount coupon functionality to the system, and eventually you write the following `apply_discount` function:

```
def apply_discount(product, discount):
    price = int(product['price'] * (1.0 - discount))
    assert 0 <= price <= product['price']
    return price
```

Notice the `assert` statement in there? It will guarantee that, no matter what, discounted prices calculated by this function cannot be lower

than \$0 and they cannot be higher than the original price of the product.

Let's make sure this actually works as intended if we call this function to apply a valid discount. In this example, products for our store will be represented as plain dictionaries. This is probably not what you'd do for a real application, but it'll work nicely for demonstrating assertions. Let's create an example product—a pair of nice shoes at a price of \$149.00:

```
>>> shoes = {'name': 'Fancy Shoes', 'price': 14900}
```

By the way, did you notice how I avoided currency rounding issues by using an integer to represent the price amount in cents? That's generally a good idea... But I digress. Now, if we apply a 25% discount to these shoes, we would expect to arrive at a sale price of \$111.75:

```
>>> apply_discount(shoes, 0.25)
11175
```

Alright, this worked nicely. Now, let's try to apply some invalid discounts. For example, a 200% “discount” that would lead to us giving money to the customer:

```
>>> apply_discount(shoes, 2.0)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    apply_discount(prod, 2.0)
  File "<input>", line 4, in apply_discount
    assert 0 <= price <= product['price']
AssertionError
```

As you can see, when we try to apply this invalid discount, our program halts with an `AssertionError`. This happens because a discount of 200% violated the assertion condition we placed in the `apply_discount` function.

You can also see how the exception stacktrace points out the exact line of code containing the failed assertion. If you (or another developer on your team) ever encounter one of these errors while testing the online store, it will be easy to find out what happened just by looking at the exception traceback.

This speeds up debugging efforts considerably, and it will make your programs more maintainable in the long-run. And that, my friend, is the power of assertions.

### Why Not Just Use a Regular Exception?

Now, you're probably wondering why I didn't just use an if-statement and an exception in the previous example...

You see, the proper use of assertions is to inform developers about *unrecoverable* errors in a program. Assertions are *not* intended to signal expected error conditions, like a File-Not-Found error, where a user can take corrective actions or just try again.

Assertions are meant to be *internal self-checks* for your program. They work by declaring some conditions as *impossible* in your code. If one of these conditions doesn't hold, that means there's a bug in the program.

If your program is bug-free, these conditions will never occur. But if they *do* occur, the program will crash with an assertion error telling you exactly which “impossible” condition was triggered. This makes it much easier to track down and fix bugs in your programs. And I like anything that makes life easier—don't you?

For now, keep in mind that Python's `assert` statement is a debugging aid, not a mechanism for handling run-time errors. The goal of using assertions is to let developers find the likely root cause of a bug more quickly. An assertion error should never be raised unless there's a bug in your program.

Let's take a closer look at some other things we can do with assertions,



and then I'll cover two common pitfalls when using them in real-world scenarios.

## Python's Assert Syntax

It's always a good idea to study up on how a language feature is actually implemented in Python before you start using it. So let's take a quick look at the syntax for the assert statement, according to the Python docs:<sup>1</sup>

```
assert_stmt ::= "assert" expression1 ["," expression2]
```

In this case, `expression1` is the condition we test, and the optional `expression2` is an error message that's displayed if the assertion fails. At execution time, the Python interpreter transforms each assert statement into roughly the following sequence of statements:

```
if __debug__:
    if not expression1:
        raise AssertionError(expression2)
```

Two interesting things about this code snippet:

Before the assert condition is checked, there's an additional check for the `__debug__` global variable. It's a built-in boolean flag that's true under normal circumstances and false if optimizations are requested. We'll talk some more about later that in the "common pitfalls" section.

Also, you can use `expression2` to pass an optional error message that will be displayed with the `AssertionError` in the traceback. This can simplify debugging even further. For example, I've seen code like this:

```
>>> if cond == 'x':
...     do_x()
```

---

<sup>1</sup>cf. Python Docs: "The Assert Statement"

```
... elif cond == 'y':  
...     do_y()  
... else:  
...     assert False, (  
...         'This should never happen, but it does '  
...         'occasionally. We are currently trying to '  
...         'figure out why. Email dbader if you '  
...         'encounter this in the wild. Thanks!')
```

Is this ugly? Well, yes. But it’s definitely a valid and helpful technique if you’re faced with a Heisenbug<sup>2</sup> in one of your applications.

## Common Pitfalls With Using Asserts in Python

Before you move on, there are two important caveats regarding the use of assertions in Python that I’d like to call out.

The first one has to do with introducing security risks and bugs into your applications, and the second one is about a syntax quirk that makes it easy to write *useless* assertions.

This sounds (and potentially is) quite horrible, so you should probably at least skim these two caveats below.

### Caveat #1 – Don’t Use Asserts for Data Validation

The biggest caveat with using asserts in Python is that assertions can be globally disabled<sup>3</sup> with the `-O` and `-OO` command line switches, as well as the `PYTHONOPTIMIZE` environment variable in CPython.

This turns any assert statement into a null-operation: the assertions simply get compiled away and won’t be evaluated, which means that none of the conditional expressions will be executed.

---

<sup>2</sup>cf. [Wikipedia: Heisenbug](#)

<sup>3</sup>cf. [Python Docs: “Constants \(\\_\\_debug\\_\\_\)”](#)

This is an intentional design decision used similarly by many other programming languages. As a side-effect, it becomes extremely dangerous to use `assert` statements as a quick and easy way to validate input data.

Let me explain—if your program uses asserts to check if a function argument contains a “wrong” or unexpected value, this can backfire quickly and lead to bugs or security holes.

Let’s take a look at a simple example that demonstrates this problem. Again, imagine you’re building an online store application with Python. Somewhere in your application code there’s a function to delete a product as per a user’s request.

Because you just learned about assertions, you’re eager to use them in your code (hey, I know I would be!) and you write the following implementation:

```
def delete_product(prod_id, user):  
    assert user.is_admin(), 'Must be admin'  
    assert store.has_product(prod_id), 'Unknown product'  
    store.get_product(prod_id).delete()
```

Take a close look at this `delete_product` function. Now, what’s going to happen if assertions are disabled?

There are two serious issues in this three-line function example, and they’re caused by the incorrect use of `assert` statements:

1. **Checking for admin privileges with an `assert` statement is dangerous.** If assertions are disabled in the Python interpreter, this turns into a null-op. Therefore *any user can now delete products*. The privileges check doesn’t even run. This likely introduces a security problem and opens the door for attackers to destroy or severely damage the data in our online store. Not good.

2. **The `has_product()` check is skipped when assertions are disabled.** This means `get_product()` can now be called with invalid product IDs—which could lead to more severe bugs, depending on how our program is written. In the worst case, this could be an avenue for someone to launch Denial of Service attacks against our store. For example, if the store app crashes if someone attempts to delete an unknown product, an attacker could bombard it with invalid delete requests and cause an outage.

How might we avoid these problems? The answer is to *never* use assertions to do data validation. Instead, we could do our validation with regular `if`-statements and raise validation exceptions if necessary, like so:

```
def delete_product(product_id, user):
    if not user.is_admin():
        raise AuthError('Must be admin to delete')
    if not store.has_product(product_id):
        raise ValueError('Unknown product id')
    store.get_product(product_id).delete()
```

This updated example also has the benefit that instead of raising un-specific `AssertionError` exceptions, it now raises semantically correct exceptions like `ValueError` or `AuthError` (which we'd have to define ourselves.)

## Caveat #2 – Asserts That Never Fail

It's surprisingly easy to accidentally write Python `assert` statements that always evaluate to `true`. I've been bitten by this myself in the past. Here's the problem, in a nutshell:

When you pass a tuple as the first argument in an `assert` statement, the assertion always evaluates as `true` and therefore never fails.

For example, this assertion will never fail:

```
assert(1 == 2, 'This should fail')
```

This has to do with non-empty tuples always being truthy in Python. If you pass a tuple to an assert statement, it leads to the assert condition always being true—which in turn leads to the above assert statement being *useless* because it can never fail and trigger an exception.

It's relatively easy to accidentally write bad multi-line asserts due to this, well, unintuitive behavior. For example, I merrily wrote a bunch of broken test cases that gave a false sense of security in one of my test suites. Imagine you had this assertion in one of your unit tests:

```
assert (  
    counter == 10,  
    'It should have counted all the items'  
)
```

Upon first inspection, this test case looks completely fine. However, it would never catch an incorrect result: the assertion always evaluates to True, regardless of the state of the counter variable. And why is that? Because it asserts the truth value of a tuple object.

Like I said, it's rather easy to shoot yourself in the foot with this (mine still hurts). A good countermeasure you can apply to prevent this syntax quirk from causing trouble is to use a code linter.<sup>4</sup> Newer versions of Python 3 will also show a syntax warning for these dubious asserts.

By the way, that's also why you should always do a quick smoke test with your unit test cases. Make sure they can actually fail before you move on to writing the next one.

---

<sup>4</sup>I wrote an article about avoiding bogus assertions in your Python tests. You can find it here: [dbader.org/blog/catching-bogus-python-asserts](http://dbader.org/blog/catching-bogus-python-asserts).

### Python Assertions — Summary

Despite these caveats I believe that Python's assertions are a powerful debugging tool that's frequently underused by Python developers.

Understanding how assertions work and when to apply them can help you write Python programs that are more maintainable and easier to debug.

It's a great skill to learn that will help bring your Python knowledge to the next level and make you a more well-rounded Pythonista. I know it has saved me hours upon hours of debugging.

### Key Takeaways

- Python's `assert` statement is a debugging aid that tests a condition as an internal self-check in your program.
- Asserts should only be used to help developers identify bugs. They're not a mechanism for handling run-time errors.
- Asserts can be globally disabled with an interpreter setting.

## 2.2 Complacent Comma Placement

Here's a handy tip for when you're adding and removing items from a list, dict, or set constant in Python: Just end all of your lines with a comma.

Not sure what I'm talking about? Let me give you a quick example. Imagine you've got this list of names in your code:

```
>>> names = ['Alice', 'Bob', 'Dilbert']
```

Whenever you make a change to this list of names, it'll be hard to tell what was modified by looking at a Git diff, for example. Most source control systems are line-based and have a hard time highlighting multiple changes to a single line.

A quick fix for that is to adopt a code style where you spread out list, dict, or set constants across multiple lines, like so:

```
>>> names = [  
...     'Alice',  
...     'Bob',  
...     'Dilbert'  
... ]
```

That way there's one item per line, making it perfectly clear which one was added, removed, or modified when you view a diff in your source control system. It's a small change but I found it helped me avoid silly mistakes. It also made it easier for my teammates to review my code changes.

Now, there are two editing cases that can still cause some confusion. Whenever you add a new item at the end of a list, or you remove the last item, you'll have to update the comma placement manually to get consistent formatting.

Let's say you'd like to add another name (*Jane*) to that list. If you add *Jane*, you'll need to fix the comma placement after the *Dilbert* line to avoid a nasty error:

```
>>> names = [  
...     'Alice',  
...     'Bob',  
...     'Dilbert' # <- Missing comma!  
...     'Jane'  
]
```

When you inspect the contents of that list, brace yourself for a surprise:

```
>>> names  
['Alice', 'Bob', 'DilbertJane']
```

As you can see, Python *merged* the strings *Dilbert* and *Jane* into *DilbertJane*. This so-called “string literal concatenation” is intentional and documented behavior. And it's also a fantastic way to shoot yourself in the foot by introducing hard-to-catch bugs into your programs:

“Multiple adjacent string or bytes literals (delimited by whitespace), possibly using different quoting conventions, are allowed, and their meaning is the same as their concatenation.”<sup>5</sup>

Still, string literal concatenation is a useful feature in some cases. For example, you can use it to reduce the number of backslashes needed to split long string constants across multiple lines:

---

<sup>5</sup>cf. Python Docs: “String literal concatenation”



```
my_str = ('This is a super long string constant '  
          'spread out across multiple lines. '  
          'And look, no backslash characters needed!')
```

On the other hand, we've just seen how the same feature can quickly turn into a liability. Now, how do we fix this situation?

Adding the missing comma after *Dilbert* prevents the two strings from getting merged into one:

```
>>> names = [  
...     'Alice',  
...     'Bob',  
...     'Dilbert',  
...     'Jane'  
]
```

But now we've come full circle and returned to the original problem. I had to modify two lines in order to add a new name to the list. This makes it harder to see what was modified in the Git diff again... Did someone add a new name? Did someone change Dilbert's name?

Luckily, Python's syntax allows for some leeway to solve this comma placement issue once and for all. You just need to train yourself to adopt a code style that avoids it in the first place. Let me show you how.

In Python, you can place a comma after every item in a list, dict, or set constant, including the last item. That way, you can just remember to always end your lines with a comma and thus avoid the comma placement juggling that would otherwise be required.

Here's what the final example looks like:

```
>>> names = [  
...     'Alice',
```

```
...     'Bob',  
...     'Dilbert',  
... ]
```

Did you spot the comma after *Dilbert*? That'll make it easy to add or remove new items without having to update the comma placement. It keeps your lines consistent, your source control diffs clean, and your code reviewers happy. Hey, sometimes the magic is in the little things, right?

### Key Takeaways

- Smart formatting and comma placement can make your list, dict, or set constants easier to maintain.
- Python's string literal concatenation feature can work to your benefit, or introduce hard-to-catch bugs.

## 2.3 Context Managers and the with Statement

The with statement in Python is regarded as an obscure feature by some. But when you peek behind the scenes, you'll see that there's no *magic* involved, and it's actually a highly useful feature that can help you write cleaner and more readable Python code.

So what's the with statement good for? It helps simplify some common resource management patterns by abstracting their functionality and allowing them to be factored out and reused.

A good way to see this feature used effectively is by looking at examples in the Python standard library. The built-in open() function provides us with an excellent use case:

```
with open('hello.txt', 'w') as f:
    f.write('hello, world!')
```

Opening files using the with statement is generally recommended because it ensures that open file descriptors are closed automatically after program execution leaves the context of the with statement. Internally, the above code sample translates to something like this:

```
f = open('hello.txt', 'w')
try:
    f.write('hello, world')
finally:
    f.close()
```

You can already tell that this is quite a bit more verbose. Note that the try...finally statement is significant. It wouldn't be enough to just write something like this:

```
f = open('hello.txt', 'w')
f.write('hello, world')
f.close()
```

This implementation won't guarantee the file is closed if there's an exception during the `f.write()` call—and therefore our program might leak a file descriptor. That's why the `with` statement is so useful. It makes properly acquiring and releasing resources a breeze.

Another good example where the `with` statement is used effectively in the Python standard library is the `threading.Lock` class:

```
some_lock = threading.Lock()

# Harmful:
some_lock.acquire()
try:
    # Do something...
finally:
    some_lock.release()

# Better:
with some_lock:
    # Do something...
```

In both cases, using a `with` statement allows you to abstract away most of the resource handling logic. Instead of having to write an explicit `try...finally` statement each time, using the `with` statement takes care of that for us.

The `with` statement can make code that deals with system resources more readable. It also helps you avoid bugs or leaks by making it practically impossible to forget to clean up or release a resource when it's no longer needed.

## Supporting with in Your Own Objects

Now, there's nothing special or magical about the `open()` function or the `threading.Lock` class and the fact that they can be used with a `with` statement. You can provide the same functionality in your own classes and functions by implementing so-called *context managers*.<sup>6</sup>

What's a context manager? It's a simple “protocol” (or interface) that your object needs to follow in order to support the `with` statement. Basically, all you need to do is add `__enter__` and `__exit__` methods to an object if you want it to function as a context manager. Python will call these two methods at the appropriate times in the resource management cycle.

Let's take a look at what this would look like in practical terms. Here's what a simple implementation of the `open()` context manager might look like:

```
class ManagedFile:
    def __init__(self, name):
        self.name = name

    def __enter__(self):
        self.file = open(self.name, 'w')
        return self.file

    def __exit__(self, exc_type, exc_val, exc_tb):
        if self.file:
            self.file.close()
```

Our `ManagedFile` class follows the context manager protocol and now supports the `with` statement, just like the original `open()` example did:

---

<sup>6</sup>cf. Python Docs: “With Statement Context Managers”

```
>>> with ManagedFile('hello.txt') as f:
...     f.write('hello, world!')
...     f.write('bye now')
```

Python calls `__enter__` when execution *enters* the context of the with statement and it's time to acquire the resource. When execution *leaves* the context again, Python calls `__exit__` to free up the resource.

Writing a class-based context manager isn't the only way to support the with statement in Python. The `contextlib`<sup>7</sup> utility module in the standard library provides a few more abstractions built on top of the basic context manager protocol. This can make your life a little easier if your use cases match what's offered by `contextlib`.

For example, you can use the `contextlib.contextmanager` decorator to define a generator-based *factory function* for a resource that will then automatically support the with statement. Here's what rewriting our `ManagedFile` context manager example with this technique looks like:

```
from contextlib import contextmanager

@contextmanager
def managed_file(name):
    try:
        f = open(name, 'w')
        yield f
    finally:
        f.close()

>>> with managed_file('hello.txt') as f:
...     f.write('hello, world!')
...     f.write('bye now')
```

---

<sup>7</sup>cf. Python Docs: “`contextlib`”

In this case, `managed_file()` is a generator that first acquires the resource. After that, it temporarily suspends its own execution and *yields* the resource so it can be used by the caller. When the caller leaves the `with` context, the generator continues to execute so that any remaining clean-up steps can occur and the resource can get released back to the system.

The class-based implementation and the generator-based one are essentially equivalent. You might prefer one over the other, depending on which approach you find more readable.

A downside of the `@contextmanager`-based implementation might be that it requires some understanding of advanced Python concepts like decorators and generators. If you need to get up to speed with those, feel free to take a detour to the relevant chapters here in this book.

Once again, making the right implementation choice here comes down to what you and your team are comfortable using and what you find the most readable.

### Writing Pretty APIs With Context Managers

Context managers are quite flexible, and if you use the `with` statement creatively, you can define convenient APIs for your modules and classes.

For example, what if the “resource” we wanted to manage was text indentation levels in some kind of report generator program? What if we could write code like this to do it:

```
with Indenter() as indent:
    indent.print('hi!')
    with indent:
        indent.print('hello')
        with indent:
            indent.print('bonjour')
    indent.print('hey')
```

This almost reads like a domain-specific language (DSL) for indenting text. Also, notice how this code enters and leaves the same context manager multiple times to change indentation levels. Running this code snippet should lead to the following output and print neatly formatted text to the console:

```
hi!  
    hello  
        bonjour  
hey
```

So, how would you implement a context manager to support this functionality?

By the way, this could be a great exercise for you to understand exactly how context managers work. So before you check out my implementation below, you might want to take some time and try to implement this yourself as a learning exercise.

If you're ready to check out my implementation, here's how you might implement this functionality using a class-based context manager:

```
class Indenter:  
    def __init__(self):  
        self.level = 0  
  
    def __enter__(self):  
        self.level += 1  
        return self  
  
    def __exit__(self, exc_type, exc_val, exc_tb):  
        self.level -= 1  
  
    def print(self, text):  
        print('    ' * self.level + text)
```



That wasn't so bad, was it? I hope that by now you're already feeling more comfortable using context managers and the `with` statement in your own Python programs. They're an excellent feature that will allow you to deal with resource management in a much more Pythonic and maintainable way.

If you're looking for another exercise to deepen your understanding, try implementing a context manager that measures the execution time of a code block using the `time.time` function. Be sure to try out writing both a decorator-based and a class-based variant to drive home the difference between the two.

### Key Takeaways

- The `with` statement simplifies exception handling by encapsulating standard uses of `try/finally` statements in so-called context managers.
- Most commonly it is used to manage the safe acquisition and release of system resources. Resources are acquired by the `with` statement and released automatically when execution leaves the `with` context.
- Using `with` effectively can help you avoid resource leaks and make your code easier to read.

## 2.4 Underscores, Dunders, and More

Single and double underscores have a meaning in Python variable and method names. Some of that meaning is merely by convention and intended as a hint to the programmer—and some of it is enforced by the Python interpreter.

If you're wondering, *“What’s the meaning of single and double underscores in Python variable and method names?”* I’ll do my best to get you the answer here. In this chapter we’ll discuss the following five underscore patterns and naming conventions, and how they affect the behavior of your Python programs:

- Single Leading Underscore: `_var`
- Single Trailing Underscore: `var_`
- Double Leading Underscore: `__var`
- Double Leading and Trailing Underscore: `__var__`
- Single Underscore: `_`

### 1. Single Leading Underscore: “\_var”

When it comes to variable and method names, the single underscore prefix has a meaning by convention only. It’s a hint to the programmer—it means what the Python community agrees it should mean, but it does not affect the behavior of your programs.

The underscore prefix is meant as a *hint* to tell another programmer that a variable or method starting with a single underscore is intended for internal use. This convention is defined in PEP 8, the most commonly used Python code style guide.<sup>8</sup>

However, this convention isn’t enforced by the Python interpreter. Python does not have strong distinctions between “private” and “public” variables like Java does. Adding a single underscore in front of a variable name is more like someone putting up a tiny underscore

---

<sup>8</sup>cf. PEP 8: “Style Guide for Python Code”

warning sign that says: *“Hey, this isn’t really meant to be a part of the public interface of this class. Best to leave it alone.”*

Take a look at the following example:

```
class Test:
    def __init__(self):
        self.foo = 11
        self._bar = 23
```

What’s going to happen if you instantiate this class and try to access the `foo` and `_bar` attributes defined in its `__init__` constructor?

Let’s find out:

```
>>> t = Test()
>>> t.foo
11
>>> t._bar
23
```

As you can see, the leading single underscore in `_bar` did not prevent us from “reaching into” the class and accessing the value of that variable.

That’s because the single underscore prefix in Python is merely an agreed-upon convention—at least when it comes to variable and method names. However, leading underscores do impact how names get imported from modules. Imagine you had the following code in a module called `my_module`:

```
# my_module.py:

def external_func():
    return 23
```

```
def _internal_func():  
    return 42
```

Now, if you use a *wildcard import* to import all the names from the module, Python will *not* import names with a leading underscore (unless the module defines an `__all__` list that overrides this behavior<sup>9</sup>):

```
>>> from my_module import *  
>>> external_func()  
23  
>>> _internal_func()  
NameError: "name '_internal_func' is not defined"
```

By the way, wildcard imports should be avoided as they make it unclear which names are present in the namespace.<sup>10</sup> It's better to stick to regular imports for the sake of clarity. Unlike wildcard imports, regular imports are not affected by the leading single underscore naming convention:

```
>>> import my_module  
>>> my_module.external_func()  
23  
>>> my_module._internal_func()  
42
```

I know this might be a little confusing at this point. If you stick to the PEP 8 recommendation that wildcard imports should be avoided, then all you really need to remember is this:

Single underscores are a Python naming convention that indicates a name is meant for internal use. It is generally not enforced by the Python interpreter and is only meant as a hint to the programmer.

---

<sup>9</sup>cf. Python Docs: “Importing \* From a Package”

<sup>10</sup>cf. PEP 8: “Imports”

## 2. Single Trailing Underscore: “var\_”

Sometimes the most fitting name for a variable is already taken by a keyword in the Python language. Therefore, names like `class` or `def` cannot be used as variable names in Python. In this case, you can append a single underscore to break the naming conflict:

```
>>> def make_object(name, class):  
SyntaxError: "invalid syntax"  
  
>>> def make_object(name, class_):  
...     pass
```

In summary, a single trailing underscore (postfix) is used by convention to avoid naming conflicts with Python keywords. This convention is defined and explained in PEP 8.

## 3. Double Leading Underscore: “\_\_var”

The naming patterns we’ve covered so far receive their meaning from agreed-upon conventions only. With Python class attributes (variables and methods) that start with double underscores, things are a little different.

A double underscore prefix causes the Python interpreter to rewrite the attribute name in order to avoid naming conflicts in subclasses.

This is also called *name mangling*—the interpreter changes the name of the variable in a way that makes it harder to create collisions when the class is extended later.

I know this sounds rather abstract. That’s why I put together this little code example we can use for experimentation:

```
class Test:  
    def __init__(self):  
        self.foo = 11
```

```
self._bar = 23
self.__baz = 23
```

Let's take a look at the attributes on this object using the built-in `dir()` function:

```
>>> t = Test()
>>> dir(t)
['_Test__baz', '__class__', '__delattr__', '__dict__',
 '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattr__', '__gt__', '__hash__', '__init__',
 '__le__', '__lt__', '__module__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__', '__weakref__', '_bar', 'foo']
```

This gives us a list with the object's attributes. Let's take this list and look for our original variable names `foo`, `_bar`, and `__baz`. I promise you'll notice some interesting changes.

First of all, the `self.foo` variable appears unmodified as `foo` in the attribute list.

Next up, `self._bar` behaves the same way—it shows up on the class as `_bar`. Like I said before, the leading underscore is just a *convention* in this case—a hint for the programmer.

However, with `self.__baz` things look a little different. When you search for `__baz` in that list, you'll see that there is no variable with that name.

So what happened to `__baz`?

If you look closely, you'll see there's an attribute called `_Test__baz` on this object. This is the *name mangling* that the Python interpreter applies. It does this to protect the variable from getting overridden in subclasses.

Let's create another class that extends the Test class and attempts to override its existing attributes added in the constructor:

```
class ExtendedTest(Test):
    def __init__(self):
        super().__init__()
        self.foo = 'overridden'
        self._bar = 'overridden'
        self.__baz = 'overridden'
```

Now, what do you think the values of `foo`, `_bar`, and `__baz` will be on instances of this `ExtendedTest` class? Let's take a look:

```
>>> t2 = ExtendedTest()
>>> t2.foo
'overridden'
>>> t2._bar
'overridden'
>>> t2.__baz
AttributeError:
"'ExtendedTest' object has no attribute '__baz'"
```

Wait, why did we get that `AttributeError` when we tried to inspect the value of `t2.__baz`? Name mangling strikes again! It turns out this object doesn't even have a `__baz` attribute:

```
>>> dir(t2)
['_ExtendedTest__baz', '_Test__baz', '__class__',
 '__delattr__', '__dict__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattr__',
 '__gt__', '__hash__', '__init__', '__le__', '__lt__',
 '__module__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__',
 '__weakref__', '_bar', 'foo', 'get_vars']
```

As you can see, `__baz` got turned into `_ExtendedTest__baz` to prevent accidental modification. But the original `_Test__baz` is also still around:

```
>>> t2._ExtendedTest__baz
'overridden'
>>> t2._Test__baz
42
```

Double underscore name mangling is fully transparent to the programmer. Take a look at the following example that will confirm this:

```
class ManglingTest:
    def __init__(self):
        self.__mangled = 'hello'

    def get_mangled(self):
        return self.__mangled

>>> ManglingTest().get_mangled()
'hello'
>>> ManglingTest().__mangled
AttributeError:
"'ManglingTest' object has no attribute '__mangled'"
```

Does name mangling also apply to method names? It sure does! Name mangling affects *all* names that start with two underscore characters (“dunders”) in a class context:

```
class MangledMethod:
    def __method(self):
        return 42

    def call_it(self):
```



```
        return self.__method()

>>> MangledMethod().__method()
AttributeError:
"'MangledMethod' object has no attribute '__method'"
>>> MangledMethod().call_it()
42
```

Here's another, perhaps surprising, example of name mangling in action:

```
_MangledGlobal__mangled = 23

class MangledGlobal:
    def test(self):
        return __mangled

>>> MangledGlobal().test()
23
```

In this example, I declared `_MangledGlobal__mangled` as a global variable. Then I accessed the variable inside the context of a class named `MangledGlobal`. Because of name mangling, I was able to reference the `_MangledGlobal__mangled` global variable as just `__mangled` inside the `test()` method on the class.

The Python interpreter automatically expanded the name `__mangled` to `_MangledGlobal__mangled` because it begins with two underscore characters. This demonstrates that name mangling isn't tied to class attributes specifically. It applies to any name starting with two underscore characters that is used in a class context.

Whew! That was a lot to absorb.

To be honest with you, I didn't write down these examples and explanations off the top of my head. It took me some research and editing

to do it. I've been using Python for years but rules and special cases like that aren't constantly on my mind.

Sometimes the most important skills for a programmer are “pattern recognition” and knowing where to look things up. If you feel a little overwhelmed at this point, don't worry. Take your time and play with some of the examples in this chapter.

Let these concepts sink in enough so that you'll recognize the general idea of name mangling and some of the other behaviors I've shown you. If you encounter them “in the wild” one day, you'll know what to look for in the documentation.

### **Sidebar: What are *dunders*?**

If you've heard some experienced Pythonistas talk about Python or watched a few conference talks you may have heard the term *dunder*. If you're wondering what that is, well, here's your answer:

Double underscores are often referred to as “dunders” in the Python community. The reason is that double underscores appear quite often in Python code, and to avoid fatiguing their jaw muscles, Pythonistas often shorten “double underscore” to “dunder.”

For example, you'd pronounce `__baz` as “dunder baz.” Likewise, `__init__` would be pronounced as “dunder init,” even though one might think it should be “dunder init dunder.”

But that's just yet another quirk in the naming convention. It's like a *secret handshake* for Python developers.

## **4. Double Leading and Trailing Underscore:**

### **“`__var__`”**

Perhaps surprisingly, name mangling is *not* applied if a name *starts and ends* with double underscores. Variables surrounded by a double underscore prefix and postfix are left unscathed by the Python interpreter:

```
class PrefixPostfixTest:
    def __init__(self):
        self.__bam__ = 42

>>> PrefixPostfixTest().__bam__
42
```

However, names that have both leading and trailing double underscores are reserved for special use in the language. This rule covers things like `__init__` for object constructors, or `__call__` to make objects callable.

These *dunder methods* are often referred to as *magic methods*—but many people in the Python community, including myself, don’t like that word. It implies that the use of dunder methods is discouraged, which is entirely not the case. They’re a core feature in Python and should be used as needed. There’s nothing “magical” or arcane about them.

However, as far as naming conventions go, it’s best to stay away from using names that start and end with double underscores in your own programs to avoid collisions with future changes to the Python language.

## 5. Single Underscore: “\_”

Per convention, a single stand-alone underscore is sometimes used as a name to indicate that a variable is temporary or insignificant.

For example, in the following loop we don’t need access to the running index and we can use “\_” to indicate that it is just a temporary value:

```
>>> for _ in range(32):
...     print('Hello, World.')
```

You can also use single underscores in unpacking expressions as a

“don’t care” variable to ignore particular values. Again, this meaning is per convention only and it doesn’t trigger any special behaviors in the Python parser. The single underscore is simply a valid variable name that’s sometimes used for this purpose.

In the following code example, I’m unpacking a tuple into separate variables but I’m only interested in the values for the `color` and `mileage` fields. However, in order for the unpacking expression to succeed, I need to assign all values contained in the tuple to variables. That’s where “`_`” is useful as a placeholder variable:

```
>>> car = ('red', 'auto', 12, 3812.4)
>>> color, _, _, mileage = car

>>> color
'red'
>>> mileage
3812.4
>>> _
12
```

Besides its use as a temporary variable, “`_`” is a special variable in most Python REPLs that represents the result of the last expression evaluated by the interpreter.

This is handy if you’re working in an interpreter session and you’d like to access the result of a previous calculation:

```
>>> 20 + 3
23
>>> _
23
>>> print(_)
23
```

It’s also handy if you’re constructing objects on the fly and want to interact with them without assigning them a name first:

```
>>> list()
[]
>>> _.append(1)
>>> _.append(2)
>>> _.append(3)
>>> _
[1, 2, 3]
```

## Key Takeaways

- **Single Leading Underscore** “\_var”: Naming convention indicating a name is meant for internal use. Generally not enforced by the Python interpreter (except in wildcard imports) and meant as a hint to the programmer only.
- **Single Trailing Underscore** “var\_”: Used by convention to avoid naming conflicts with Python keywords.
- **Double Leading Underscore** “\_\_var”: Triggers name mangling when used in a class context. Enforced by the Python interpreter.
- **Double Leading and Trailing Underscore** “\_\_var\_\_”: Indicates special methods defined by the Python language. Avoid this naming scheme for your own attributes.
- **Single Underscore** “\_”: Sometimes used as a name for temporary or insignificant variables (“don’t care”). Also, it represents the result of the last expression in a Python REPL session.

## 2.5 A Shocking Truth About String Formatting

Remember the Zen of Python and how there should be “one obvious way to do something?” You might scratch your head when you find out that there are *four* major ways to do string formatting in Python.

In this chapter I’ll demonstrate how these four string formatting approaches work and what their respective strengths and weaknesses are. I’ll also give you my simple “rule of thumb” for how I pick the best general-purpose string formatting approach.

Let’s jump right in, as we’ve got a lot to cover. In order to have a simple toy example for experimentation, let’s assume we’ve got the following variables (or constants, really) to work with:

```
>>> errno = 50159747054
>>> name = 'Bob'
```

And based on these variables we’d like to generate an output string with the following error message:

```
'Hey Bob, there is a 0xbadc0ffee error!'
```

Now, *that* error could really spoil a dev’s Monday morning! But we’re here to discuss string formatting today. So let’s get to work.

### #1 – “Old Style” String Formatting

Strings in Python have a unique built-in operation that can be accessed with the %-operator. It’s a shortcut that lets you do simple positional formatting very easily. If you’ve ever worked with a `printf`-style function in C, you’ll instantly recognize how this works. Here’s a simple example:

```
>>> 'Hello, %s' % name
'Hello, Bob'
```

I'm using the `%s` format specifier here to tell Python where to substitute the value of `name`, represented as a string. This is called “old style” string formatting.

In old style string formatting there are also other format specifiers available that let you control the output string. For example, it's possible to convert numbers to hexadecimal notation or to add whitespace padding to generate nicely formatted tables and reports.<sup>11</sup>

Here, I'm using the `%x` format specifier to convert an int value to a string and to represent it as a hexadecimal number:

```
>>> '%x' % errno
'badc0ffee'
```

The “old style” string formatting syntax changes slightly if you want to make multiple substitutions in a single string. Because the `%`-operator only takes one argument, you need to wrap the right-hand side in a tuple, like so:

```
>>> 'Hey %s, there is a 0x%x error!' % (name, errno)
'Hey Bob, there is a 0xbadc0ffee error!'
```

It's also possible to refer to variable substitutions by name in your format string, if you pass a mapping to the `%`-operator:

```
>>> 'Hey %(name)s, there is a 0x%(errno)x error!' % {
...     "name": name, "errno": errno }
'Hey Bob, there is a 0xbadc0ffee error!'
```

---

<sup>11</sup>cf. Python Docs: “printf-style String Formatting”

This makes your format strings easier to maintain and easier to modify in the future. You don't have to worry about making sure the order you're passing in the values matches up with the order the values are referenced in the format string. Of course, the downside is that this technique requires a little more typing.

I'm sure you've been wondering why this `printf`-style formatting is called “old style” string formatting. Well, let me tell you. It was technically superseded by “new style” formatting, which we're going to talk about in a minute. But while “old style” formatting has been de-emphasized, it hasn't been deprecated. It is still supported in the latest versions of Python.

## #2 – “New Style” String Formatting

Python 3 introduced a new way to do string formatting that was also later back-ported to Python 2.7. This “new style” string formatting gets rid of the %-operator special syntax and makes the syntax for string formatting more regular. Formatting is now handled by calling a `format()` function on a string object.<sup>12</sup>

You can use the `format()` function to do simple positional formatting, just like you could with “old style” formatting:

```
>>> 'Hello, {}'.format(name)
'Hello, Bob'
```

Or, you can refer to your variable substitutions by name and use them in any order you want. This is quite a powerful feature as it allows for re-arranging the order of display without changing the arguments passed to the format function:

```
>>> 'Hey {name}, there is a 0x{errno:x} error!'.format(
...     name=name, errno=errno)
'Hey Bob, there is a 0xbadc0ffee error!'
```

---

<sup>12</sup>cf. Python Docs: “[str.format\(\)](#)”



This also shows that the syntax to format an int variable as a hexadecimal string has changed. Now we need to pass a *format spec* by adding a “:x” suffix after the variable name.

Overall, the format string syntax has become more powerful without complicating the simpler use cases. It pays off to read up on this *string formatting mini-language* in the Python documentation.<sup>13</sup>

In Python 3, this “new style” string formatting is preferred over %-style formatting. However, starting with Python 3.6 there’s an even better way to format your strings. I’ll tell you all about it in the next section.

### #3 – Literal String Interpolation (Python 3.6+)

Python 3.6 adds yet another way to format strings, called *Formatted String Literals*. This new way of formatting strings lets you use embedded Python expressions inside string constants. Here’s a simple example to give you a feel for the feature:

```
>>> f'Hello, {name}!'
'Hello, Bob!'
```

This new formatting syntax is powerful. Because you can embed arbitrary Python expressions, you can even do inline arithmetic with it, like this:

```
>>> a = 5
>>> b = 10
>>> f'Five plus ten is {a + b} and not {2 * (a + b)}.'
'Five plus ten is 15 and not 30.'
```

Behind the scenes, formatted string literals are a Python parser feature that converts f-strings into a series of string constants and expressions. They then get joined up to build the final string.

---

<sup>13</sup>cf. Python Docs: “Format String Syntax”

Imagine we had the following `greet()` function that contains an f-string:

```
>>> def greet(name, question):
...     return f"Hello, {name}! How's it {question}?"
...

>>> greet('Bob', 'going')
"Hello, Bob! How's it going?"
```

When we disassemble the function and inspect what's going on behind the scenes, we can see that the f-string in the function gets transformed into something similar to the following:

```
>>> def greet(name, question):
...     return ("Hello, " + name + "! How's it " +
               question + "?")
```

The real implementation is slightly faster than that because it uses the `BUILD_STRING` opcode as an optimization.<sup>14</sup> But functionally they're the same:

```
>>> import dis
>>> dis.dis(greet)
2      0 LOAD_CONST      1 ('Hello, ')
      2 LOAD_FAST      0 (name)
      4 FORMAT_VALUE   0
      6 LOAD_CONST      2 ('! How's it ')
      8 LOAD_FAST      1 (question)
     10 FORMAT_VALUE   0
     12 LOAD_CONST      3 ('?')
     14 BUILD_STRING   5
     16 RETURN_VALUE
```

---

<sup>14</sup>cf. Python 3 bug-tracker issue #27078

String literals also support the existing format string syntax of the `str.format()` method. That allows you to solve the same formatting problems we’ve discussed in the previous two sections:

```
>>> f"Hey {name}, there's a {errno:#x} error!"  
"Hey Bob, there's a 0xbadc0ffee error!"
```

Python’s new Formatted String Literals are similar to the JavaScript Template Literals added in ES2015. I think they’re quite a nice addition to the language, and I’ve already started using them in my day-to-day Python 3 work. You can learn more about Formatted String Literals in the official Python documentation.<sup>15</sup>

### #4 – Template Strings

One more technique for string formatting in Python is Template Strings. It’s a simpler and less powerful mechanism, but in some cases this might be exactly what you’re looking for.

Let’s take a look at a simple greeting example:

```
>>> from string import Template  
>>> t = Template('Hey, $name!')  
>>> t.substitute(name=name)  
'Hey, Bob!'
```

You see here that we need to import the `Template` class from Python’s built-in `string` module. Template strings are not a core language feature but they’re supplied by a module in the standard library.

Another difference is that template strings don’t allow format specifiers. So in order to get our error string example to work, we need to transform our int error number into a hex-string ourselves:

---

<sup>15</sup>cf. Python Docs: “Formatted string literals”

```
>>> templ_string = 'Hey $name, there is a $error error!'
>>> Template(templ_string).substitute(
...     name=name, error=hex(errno))
'Hey Bob, there is a 0xbadc0ffee error!'
```

That worked great but you're probably wondering when you use template strings in your Python programs. In my opinion, the best use case for template strings is when you're handling format strings generated by users of your program. Due to their reduced complexity, template strings are a safer choice.

The more complex formatting mini-languages of other string formatting techniques might introduce security vulnerabilities to your programs. For example, it's possible for format strings to access arbitrary variables in your program.

That means, if a malicious user can supply a format string they can also potentially leak secret keys and other sensible information! Here's a simple proof of concept of how this attack might be used:

```
>>> SECRET = 'this-is-a-secret'
>>> class Error:
...     def __init__(self):
...         pass
>>> err = Error()
>>> user_input = '{error.__init__.__globals__[SECRET]}'

# Uh-oh...
>>> user_input.format(error=err)
'this-is-a-secret'
```

See how the hypothetical attacker was able to extract our secret string by accessing the `__globals__` dictionary from the format string? Scary, huh! Template Strings close this attack vector, and this makes them a safer choice if you're handling format strings generated from user input:

```
>>> user_input = '${error.__init__.__globals__[SECRET]}'  
>>> Template(user_input).substitute(error=err)  
ValueError:  
"Invalid placeholder in string: line 1, col 1"
```

## Which String Formatting Method Should I Use?

I totally get that having so much choice for how to format your strings in Python can feel very confusing. This would be a good time to bust out some flowchart infographic...

But I'm not going to do that. Instead, I'll try to boil it down to the simple rule of thumb that I apply when I'm writing Python.

Here we go—you can use this rule of thumb any time you're having difficulty deciding which string formatting method to use, depending on the circumstances:

### Dan's Python String Formatting Rule of Thumb:

*If your format strings are user-supplied, use Template Strings to avoid security issues. Otherwise, use Literal String Interpolation if you're on Python 3.6+, and "New Style" String Formatting if you're not.*

## Key Takeaways

- Perhaps surprisingly, there's more than one way to handle string formatting in Python.
- Each method has its individual pros and cons. Your use case will influence which method you should use.
- If you're having trouble deciding which string formatting method to use, try my *String Formatting Rule of Thumb*.

## 2.6 “The Zen of Python” Easter Egg

I know what follows is a common sight as far as Python books go. But there’s really no way around Tim Peters’ *Zen of Python*. I’ve benefited from revisiting it over the years, and I think Tim’s words made me a better coder. Hopefully they can do the same for you.

Also, you can tell the *Zen of Python* is a big deal because it’s included as an Easter egg in the language. Just enter a Python interpreter session and run the following:

```
>>> import this
```

### The Zen of Python, by Tim Peters

Beautiful is better than ugly.  
Explicit is better than implicit.  
Simple is better than complex.  
Complex is better than complicated.  
Flat is better than nested.  
Sparse is better than dense.  
Readability counts.  
Special cases aren’t special enough to break the rules.  
Although practicality beats purity.  
Errors should never pass silently.  
Unless explicitly silenced.  
In the face of ambiguity, refuse the temptation to guess.  
There should be one—and preferably only one—obvious way to do it.  
Although that way may not be obvious at first unless you’re Dutch.  
Now is better than never.  
Although never is often better than *right* now.  
If the implementation is hard to explain, it’s a bad idea.  
If the implementation is easy to explain, it may be a good idea.  
Namespaces are one honking great idea—let’s do more of those!

# **Chapter 3**

## **Effective Functions**

## 3.1 Python's Functions Are First-Class

Python's functions are first-class objects. You can assign them to variables, store them in data structures, pass them as arguments to other functions, and even return them as values from other functions.

Grokking these concepts intuitively will make understanding advanced features in Python like lambdas and decorators much easier. It also puts you on a path towards functional programming techniques.

Over the next few pages I'll guide you through a number of examples to help you develop this intuitive understanding. The examples will build on top of each other, so you might want to read them in sequence and even to try out some of them in a Python interpreter session as you go along.

Wrapping your head around the concepts we'll be discussing here might take a little longer than you'd expect. Don't worry—that's completely normal. I've been there. You might feel like you're banging your head against the wall, and then suddenly things will “click” and fall into place when you're ready.

Throughout this chapter I'll be using this `yell` function for demonstration purposes. It's a simple toy example with easily recognizable output:

```
def yell(text):  
    return text.upper() + '!'  
  
>>> yell('hello')  
'HELLO!'
```



## Functions Are Objects

All data in a Python program is represented by objects or relations between objects.<sup>1</sup> Things like strings, lists, modules, and functions are all objects. There's nothing particularly special about functions in Python. They're also just objects.

Because the `yell` function is an *object* in Python, you can assign it to another variable, just like any other object:

```
>>> bark = yell
```

This line doesn't call the function. It takes the function object referenced by `yell` and creates a second name, `bark`, that points to it. You could now also execute the same underlying function object by calling `bark`:

```
>>> bark('woof')
'WOOF! '
```

Function objects and their names are two separate concerns. Here's more proof: You can delete the function's original name (`yell`). Since another name (`bark`) still points to the underlying function, you can still call the function through it:

```
>>> del yell

>>> yell('hello?')
NameError: "name 'yell' is not defined"

>>> bark('hey')
'HEY! '
```

---

<sup>1</sup>cf. Python Docs: "Objects, values and types"

By the way, Python attaches a string identifier to every function at creation time for debugging purposes. You can access this internal identifier with the `__name__` attribute:<sup>2</sup>

```
>>> bark.__name__  
'yell'
```

Now, while the function's `__name__` is still “yell,” that doesn't affect how you can access the function object from your code. The name identifier is merely a debugging aid. A *variable pointing to a function* and the *function itself* are really two separate concerns.

## Functions Can Be Stored in Data Structures

Since functions are first-class citizens, you can store them in data structures, just like you can with other objects. For example, you can add functions to a list:

```
>>> funcs = [bark, str.lower, str.capitalize]  
>>> funcs  
[<function yell at 0x10ff96510>,  
 <method 'lower' of 'str' objects>,  
 <method 'capitalize' of 'str' objects>]
```

Accessing the function objects stored inside the list works like it would with any other type of object:

```
>>> for f in funcs:  
...     print(f, f('hey there'))  
<function yell at 0x10ff96510> 'HEY THERE!'  
<method 'lower' of 'str' objects> 'hey there'  
<method 'capitalize' of 'str' objects> 'Hey there'
```

---

<sup>2</sup>Since Python 3.3 there's also `__qualname__` which serves a similar purpose and provides a *qualified name* string to disambiguate function and class names (cf. PEP 3155).

You can even call a function object stored in the list without first assigning it to a variable. You can do the lookup and then immediately call the resulting “disembodied” function object within a single expression:

```
>>> funcs[0]('heyho')
'HEYHO!'
```

## Functions Can Be Passed to Other Functions

Because functions are objects, you can pass them as arguments to other functions. Here's a `greet` function that formats a greeting string using the function object passed to it and then prints it:

```
def greet(func):
    greeting = func('Hi, I am a Python program')
    print(greeting)
```

You can influence the resulting greeting by passing in different functions. Here's what happens if you pass the `bark` function to `greet`:

```
>>> greet(bark)
'HI, I AM A PYTHON PROGRAM!'
```

Of course, you could also define a new function to generate a different flavor of greeting. For example, the following `whisper` function might work better if you don't want your Python programs to sound like Optimus Prime:

```
def whisper(text):
    return text.lower() + '...'

>>> greet(whisper)
'hi, i am a python program...'
```

The ability to pass function objects as arguments to other functions is powerful. It allows you to abstract away and pass around *behavior* in your programs. In this example, the `greet` function stays the same but you can influence its output by passing in different *greeting behaviors*.

Functions that can accept other functions as arguments are also called *higher-order functions*. They are a necessity for the functional programming style.

The classical example for higher-order functions in Python is the built-in `map` function. It takes a function object and an iterable, and then calls the function on each element in the iterable, yielding the results as it goes along.

Here's how you might format a sequence of greetings all at once by *mapping* the `bark` function to them:

```
>>> list(map(bark, ['hello', 'hey', 'hi']))
['HELLO!', 'HEY!', 'HI!']
```

As you saw, `map` went through the entire list and applied the `bark` function to each element. As a result, we now have a new list object with modified greeting strings.

## Functions Can Be Nested

Perhaps surprisingly, Python allows functions to be defined inside other functions. These are often called *nested functions* or *inner functions*. Here's an example:

```
def speak(text):
    def whisper(t):
        return t.lower() + '...'
    return whisper(text)

>>> speak('Hello, World')
'hello, world...'
```

Now, what's going on here? Every time you call `speak`, it defines a new inner function `whisper` and then calls it immediately after. My brain's starting to itch just a little here but, all in all, that's still relatively straightforward stuff.

Here's the kicker though—`whisper` *does not exist* outside `speak`:

```
>>> whisper('Yo')
NameError:
"name 'whisper' is not defined"

>>> speak.whisper
AttributeError:
"'function' object has no attribute 'whisper'"
```

But what if you really wanted to access that nested `whisper` function from outside `speak`? Well, functions are objects—you can *return* the inner function to the caller of the parent function.

For example, here's a function defining two inner functions. Depending on the argument passed to top-level function, it selects and returns one of the inner functions to the caller:

```
def get_speak_func(volume):
    def whisper(text):
        return text.lower() + '...'
    def yell(text):
        return text.upper() + '!'
    if volume > 0.5:
        return yell
    else:
        return whisper
```

Notice how `get_speak_func` doesn't actually *call* any of its inner functions—it simply selects the appropriate inner function based on the `volume` argument and then returns the function object:

```
>>> get_speak_func(0.3)
<function get_speak_func.<locals>.whisper at 0x10ae18>

>>> get_speak_func(0.7)
<function get_speak_func.<locals>.yell at 0x1008c8>
```

Of course, you could then go on and call the returned function, either directly or by assigning it to a variable name first:

```
>>> speak_func = get_speak_func(0.7)
>>> speak_func('Hello')
'HELLO! '
```

Let that sink in for a second here... This means not only can functions *accept behaviors* through arguments but they can also *return behaviors*. How cool is that?

You know what, things are starting to get a little loopy here. I'm going to take a quick coffee break before I continue writing (and I suggest you do the same).

## Functions Can Capture Local State

You just saw how functions can contain inner functions, and that it's even possible to return these (otherwise hidden) inner functions from the parent function.

Best put on your seat belt now because it's going to get a little crazier still—we're about to enter even deeper functional programming territory. (You had that coffee break, right?)

Not only can functions return other functions, these inner functions can also *capture and carry some of the parent function's state* with them. Well, what does that mean?

I'm going to slightly rewrite the previous `get_speak_func` example to illustrate this. The new version takes a “volume” *and* a “text” argument right away to make the returned function immediately callable:

```
def get_speak_func(text, volume):
    def whisper():
        return text.lower() + '...'
    def yell():
        return text.upper() + '!'
    if volume > 0.5:
        return yell
    else:
        return whisper

>>> get_speak_func('Hello, World', 0.7)()
'HELLO, WORLD!'
```

Take a good look at the inner functions `whisper` and `yell` now. Notice how they no longer have a `text` parameter? But somehow they can still access the `text` parameter defined in the parent function. In fact, they seem to *capture* and “remember” the value of that argument.

Functions that do this are called *lexical closures* (or just *closures*, for short). A closure remembers the values from its enclosing lexical scope even when the program flow is no longer in that scope.

In practical terms, this means not only can functions *return behaviors* but they can also *pre-configure those behaviors*. Here's another bare-bones example to illustrate this idea:

```
def make_adder(n):
    def add(x):
        return x + n
    return add

>>> plus_3 = make_adder(3)
```

```
>>> plus_5 = make_adder(5)

>>> plus_3(4)
7
>>> plus_5(4)
9
```

In this example, `make_adder` serves as a *factory* to create and configure “adder” functions. Notice how the “adder” functions can still access the `n` argument of the `make_adder` function (the enclosing scope).

## Objects Can Behave Like Functions

While all functions are objects in Python, the reverse isn't true. Objects aren't functions. But they can be made *callable*, which allows you to *treat them like functions* in many cases.

If an object is callable it means you can use the round parentheses function call syntax on it and even pass in function call arguments. This is all powered by the `__call__` dunder method. Here's an example of class defining a callable object:

```
class Adder:
    def __init__(self, n):
        self.n = n

    def __call__(self, x):
        return self.n + x

>>> plus_3 = Adder(3)
>>> plus_3(4)
7
```

Behind the scenes, “calling” an object instance as a function attempts to execute the object's `__call__` method.



Of course, not all objects will be callable. That's why there's a built-in callable function to check whether an object appears to be callable or not:

```
>>> callable(plus_3)
True
>>> callable(yell)
True
>>> callable('hello')
False
```

## Key Takeaways

- Everything in Python is an object, including functions. You can assign them to variables, store them in data structures, and pass or return them to and from other functions (first-class functions.)
- First-class functions allow you to abstract away and pass around behavior in your programs.
- Functions can be nested and they can capture and carry some of the parent function's state with them. Functions that do this are called closures.
- Objects can be made callable. In many cases this allows you to treat them like functions.

## 3.2 Lambdas Are Single-Expression Functions

The `lambda` keyword in Python provides a shortcut for declaring small anonymous functions. Lambda functions behave just like regular functions declared with the `def` keyword. They can be used whenever function objects are required.

For example, this is how you'd define a simple lambda function carrying out an addition:

```
>>> add = lambda x, y: x + y
>>> add(5, 3)
8
```

You could declare the same `add` function with the `def` keyword, but it would be slightly more verbose:

```
>>> def add(x, y):
...     return x + y
>>> add(5, 3)
8
```

Now you might be wondering, “Why the big fuss about lambdas? If they're just a slightly more concise version of declaring functions with `def`, what's the big deal?”

Take a look at the following example and keep the words *function expression* in your head while you do that:

```
>>> (lambda x, y: x + y)(5, 3)
8
```

Okay, what happened here? I just used `lambda` to define an “add” function inline and then immediately called it with the arguments 5 and 3.

Conceptually, the *lambda expression* `lambda x, y: x + y` is the same as declaring a function with `def`, but just written inline. The key difference here is that I didn't have to bind the function object to a name before I used it. I simply stated the expression I wanted to compute as part of a lambda, and then immediately evaluated it by calling the lambda expression like a regular function.

Before you move on, you might want to play with the previous code example a little to really let the meaning of it sink in. I still remember this taking me awhile to wrap my head around. So don't worry about spending a few minutes in an interpreter session on this. It'll be worth it.

There's another syntactic difference between lambdas and regular function definitions. Lambda functions are restricted to a single expression. This means a lambda function can't use statements or annotations—not even a return statement.

How do you return values from lambdas then? Executing a lambda function evaluates its expression and then automatically returns the expression's result, so there's always an *implicit* return statement. That's why some people refer to lambdas as *single expression functions*.

## Lambdas You Can Use

When should you use lambda functions in your code? Technically, any time you're expected to supply a function object you can use a lambda expression. And because lambdas can be anonymous, you don't even need to assign them to a name first.

This can provide a handy and “unbureaucratic” shortcut to defining a function in Python. My most frequent use case for lambdas is writing short and concise *key funcs* for sorting iterables by an alternate key:

```
>>> tuples = [(1, 'd'), (2, 'b'), (4, 'a'), (3, 'c')]
>>> sorted(tuples, key=lambda x: x[1])
[(4, 'a'), (2, 'b'), (3, 'c'), (1, 'd')]
```

In the above example, we're sorting a list of tuples by the second value in each tuple. In this case, the lambda function provides a quick way to modify the sort order. Here's another sorting example you can play with:

```
>>> sorted(range(-5, 6), key=lambda x: x * x)
[0, -1, 1, -2, 2, -3, 3, -4, 4, -5, 5]
```

Both examples I showed you have more concise implementations in Python using the built-in `operator.itemgetter()` and `abs()` functions. But I hope you can see how using a lambda gives you much more flexibility. Want to sort a sequence by some arbitrary computed key? No problem. Now you know how to do it.

Here's another interesting thing about lambdas: Just like regular nested functions, lambdas also work as *lexical closures*.

What's a lexical closure? It's just a fancy name for a function that remembers the values from the enclosing lexical scope even when the program flow is no longer in that scope. Here's a (fairly academic) example to illustrate the idea:

```
>>> def make_adder(n):
...     return lambda x: x + n

>>> plus_3 = make_adder(3)
>>> plus_5 = make_adder(5)

>>> plus_3(4)
7
>>> plus_5(4)
9
```

In the above example, the `x + n` lambda can still access the value of `n` even though it was defined in the `make_adder` function (the enclosing scope).

Sometimes, using a lambda function instead of a nested function declared with the `def` keyword can express the programmer's intent more clearly. But to be honest, this isn't a common occurrence—at least not in the kind of code that I like to write. So let's talk a little more about that.

## But Maybe You Shouldn't...

On the one hand, I'm hoping this chapter got you interested in exploring Python's lambda functions. On the other hand, I feel like it's time to put up another caveat: Lambda functions should be used sparingly and with extraordinary care.

I know I've written my fair share of code using lambdas that looked “cool” but were actually a liability for me and my coworkers. If you're tempted to use a lambda, spend a few seconds (or minutes) to think if it is really the cleanest and most maintainable way to achieve the desired result.

For example, doing something like this to save two lines of code is just silly. Sure, technically it works and it's a nice enough “trick.” But it's also going to confuse the next gal or guy that has to ship a bugfix under a tight deadline:

```
# Harmful:
>>> class Car:
...     rev = lambda self: print('Wroom!')
...     crash = lambda self: print('Boom!')

>>> my_car = Car()
>>> my_car.crash()
'Boom!'
```

I have similar feelings about complicated `map()` or `filter()` constructs using lambdas. Usually it's much cleaner to go with a list comprehension or generator expression:

```
# Harmful:
>>> list(filter(lambda x: x % 2 == 0, range(16)))
[0, 2, 4, 6, 8, 10, 12, 14]

# Better:
>>> [x for x in range(16) if x % 2 == 0]
[0, 2, 4, 6, 8, 10, 12, 14]
```

If you find yourself doing anything remotely complex with lambda expressions, consider defining a standalone function with a proper name instead.

Saving a few keystrokes won't matter in the long run, but your colleagues (and your future self) will appreciate clean and readable code more than terse wizardry.

## Key Takeaways

- Lambda functions are single-expression functions that are not necessarily bound to a name (anonymous).
- Lambda functions can't use regular Python statements and always include an implicit return statement.
- Always ask yourself: *Would using a regular (named) function or a list comprehension offer more clarity?*

## 3.3 The Power of Decorators

At their core, Python’s decorators allow you to extend and modify the behavior of a callable (functions, methods, and classes) *without* permanently modifying the callable itself.

Any sufficiently generic functionality you can tack on to an existing class or function’s behavior makes a great use case for decoration. This includes the following:

- logging
- enforcing access control and authentication
- instrumentation and timing functions
- rate-limiting
- caching, and more

Now, why should you master the use of decorators in Python? After all, what I just mentioned sounded quite abstract, and it might be difficult to see how decorators can benefit you in your day-to-day work as a Python developer. Let me try to bring some clarity to this question by giving you a somewhat real-world example:

Imagine you’ve got 30 functions with business logic in your report-generating program. One rainy Monday morning your boss walks up to your desk and says: *“Happy Monday! Remember those TPS reports? I need you to add input/output logging to each step in the report generator. XYZ Corp needs it for auditing purposes. Oh, and I told them we can ship this by Wednesday.”*

Depending on whether or not you’ve got a solid grasp on Python’s decorators, this request will either send your blood pressure spiking or leave you relatively calm.

Without decorators you might be spending the next three days scrambling to modify each of those 30 functions and clutter them up with manual logging calls. Fun times, right?

If you do know your decorators however, you'll calmly smile at your boss and say: "*Don't worry Jim, I'll get it done by 2pm today.*"

Right after that you'll type the code for a generic `@audit_log` decorator (that's only about 10 lines long) and quickly paste it in front of each function definition. Then you'll commit your code and grab another cup of coffee...

I'm dramatizing here, but only a little. Decorators *can be* that powerful. I'd go as far as to say that understanding decorators is a milestone for any serious Python programmer. They require a solid grasp of several advanced concepts in the language, including the properties of *first-class functions*.

**I believe that the payoff for understanding how decorators work in Python can be enormous.**

Sure, decorators are relatively complicated to wrap your head around for the first time, but they're a highly useful feature that you'll often encounter in third-party frameworks and the Python standard library. Explaining decorators is also a *make or break* moment for any good Python tutorial. I'll do my best here to introduce you to them step by step.

Before you dive in however, now would be an excellent moment to refresh your memory on the properties of *first-class functions* in Python. There's a chapter on them in this book, and I would encourage you to take a few minutes to review it. The most important "first-class functions" takeaways for understanding decorators are:

- **Functions are objects**—they can be assigned to variables and passed to and returned from other functions
- **Functions can be defined inside other functions**—and a child function can capture the parent function's local state (lexical closures)

Alright, are you ready to do this? Let's get started.



## Python Decorator Basics

Now, what are decorators really? They “decorate” or “wrap” another function and let you execute code before and after the wrapped function runs.

Decorators allow you to define reusable building blocks that can change or extend the behavior of other functions. And, they let you do that without permanently modifying the wrapped function itself. The function’s behavior changes only when it’s *decorated*.

What might the implementation of a simple decorator look like? In basic terms, a decorator is *a callable that takes a callable as input and returns another callable*.

The following function has that property and could be considered the simplest decorator you could possibly write:

```
def null_decorator(func):  
    return func
```

As you can see, `null_decorator` is a callable (it’s a function), it takes another callable as its input, and it returns the same input callable without modifying it.

Let’s use it to *decorate* (or *wrap*) another function:

```
def greet():  
    return 'Hello!'  
  
greet = null_decorator(greet)  
  
>>> greet()  
'Hello!'
```

In this example, I’ve defined a `greet` function and then immediately decorated it by running it through the `null_decorator` function. I

know this doesn't look very useful yet. I mean, we specifically designed the null decorator to be useless, right? But in a moment this example will clarify how Python's special-case decorator syntax works.

Instead of explicitly calling `null_decorator` on `greet` and then reassigning the `greet` variable, you can use Python's `@` syntax for decorating a function more conveniently:

```
@null_decorator
def greet():
    return 'Hello!'

>>> greet()
'Hello!'
```

Putting an `@null_decorator` line in front of the function definition is the same as defining the function first and then running through the decorator. Using the `@` syntax is just *syntactic sugar* and a shortcut for this commonly used pattern.

Note that using the `@` syntax decorates the function immediately at definition time. This makes it difficult to access the undecorated original without brittle hacks. Therefore you might choose to decorate some functions manually in order to retain the ability to call the undecorated function as well.

## Decorators Can Modify Behavior

Now that you're a little more familiar with the decorator syntax, let's write another decorator that *actually does something* and modifies the behavior of the decorated function.

Here's a slightly more complex decorator which converts the result of the decorated function to uppercase letters:

```
def uppercase(func):  
    def wrapper():  
        original_result = func()  
        modified_result = original_result.upper()  
        return modified_result  
    return wrapper
```

Instead of simply returning the input function like the null decorator did, this uppercase decorator defines a new function on the fly (a closure) and uses it to *wrap* the input function in order to modify its behavior at call time.

The wrapper closure has access to the undecorated input function and it is free to execute additional code before and after calling the input function. (Technically, it doesn't even need to call the input function at all.)

Note how, up until now, the decorated function has never been executed. Actually calling the input function at this point wouldn't make any sense—you'll want the decorator to be able to modify the behavior of its input function when it eventually gets called.

You might want to let that sink in for a minute or two. I know how complicated this stuff can seem, but we'll get it sorted out together, I promise.

Time to see the uppercase decorator in action. What happens if you decorate the original greet function with it?

```
@uppercase  
def greet():  
    return 'Hello!'  
  
>>> greet()  
'HELLO!'
```

I hope this was the result you expected. Let's take a closer look at what just happened here. Unlike `null_decorator`, our uppercase decorator returns a *different function object* when it decorates a function:

```
>>> greet
<function greet at 0x10e9f0950>

>>> null_decorator(greet)
<function greet at 0x10e9f0950>

>>> uppercase(greet)
<function uppercase.<locals>.wrapper at 0x76da02f28>
```

And as you saw earlier, it needs to do that in order to modify the behavior of the decorated function when it finally gets called. The uppercase decorator is a function itself. And the only way to influence the “future behavior” of an input function it decorates is to replace (or *wrap*) the input function with a closure.

That's why uppercase defines and returns another function (the closure) that can then be called at a later time, run the original input function, and modify its result.

Decorators modify the behavior of a callable through a wrapper closure so you don't have to permanently modify the original. The original callable isn't permanently modified—its behavior changes only when decorated.

This let's you tack on reusable building blocks, like logging and other instrumentation, to existing functions and classes. It makes decorators such a powerful feature in Python that it's frequently used in the standard library and in third-party packages.

## A Quick Intermission

By the way, if you feel like you need a quick coffee break or a walk around the block at this point—that's totally normal. In my opinion

closures and decorators are some of the most difficult concepts to understand in Python.

Please, take your time and don't worry about figuring this out immediately. Playing through the code examples in an interpreter session one by one often helps make things sink in.

I know you can do it!

## Applying Multiple Decorators to a Function

Perhaps not surprisingly, you can apply more than one decorator to a function. This accumulates their effects and it's what makes decorators so helpful as reusable building blocks.

Here's an example. The following two decorators wrap the output string of the decorated function in HTML tags. By looking at how the tags are nested, you can see which order Python uses to apply multiple decorators:

```
def strong(func):
    def wrapper():
        return '<strong>' + func() + '</strong>'
    return wrapper

def emphasis(func):
    def wrapper():
        return '<em>' + func() + '</em>'
    return wrapper
```

Now let's take these two decorators and apply them to our greet function at the same time. You can use the regular @ syntax for that and just "stack" multiple decorators on top of a single function:

```
@strong
@emphasis
```

```
def greet():  
    return 'Hello!'
```

What output do you expect to see if you run the decorated function? Will the `@emphasis` decorator add its `<em>` tag first, or does `@strong` have precedence? Here's what happens when you call the decorated function:

```
>>> greet()  
'<strong><em>Hello!</em></strong>'
```

This clearly shows in what order the decorators were applied: from *bottom to top*. First, the input function was wrapped by the `@emphasis` decorator, and then the resulting (decorated) function got wrapped again by the `@strong` decorator.

To help me remember this bottom to top order, I like to call this behavior *decorator stacking*. You start building the stack at the bottom and then keep adding new blocks on top to work your way upwards.

If you break down the above example and avoid the `@` syntax to apply the decorators, the chain of decorator function calls looks like this:

```
decorated_greet = strong(emphasis(greet))
```

Again, you can see that the `emphasis` decorator is applied first and then the resulting wrapped function is wrapped again by the `strong` decorator.

This also means that deep levels of decorator stacking will eventually have an effect on performance because they keep adding nested function calls. In practice, this usually won't be a problem, but it's something to keep in mind if you're working on performance-intensive code that frequently uses decoration.

## Decorating Functions That Accept Arguments

All examples so far only decorated a simple *nullary* greet function that didn't take any arguments whatsoever. Up until now, the decorators you saw here didn't have to deal with forwarding arguments to the input function.

If you try to apply one of these decorators to a function that takes arguments, it will not work correctly. How do you decorate a function that takes arbitrary arguments?

This is where Python's `*args` and `**kwargs` feature<sup>3</sup> for dealing with variable numbers of arguments comes in handy. The following proxy decorator takes advantage of that:

```
def proxy(func):
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs)
    return wrapper
```

There are two notable things going on with this decorator:

- It uses the `*` and `**` operators in the wrapper closure definition to collect all positional and keyword arguments and stores them in variables (`args` and `kwargs`).
- The wrapper closure then forwards the collected arguments to the original input function using the `*` and `**` “argument unpacking” operators.

It's a bit unfortunate that the meaning of the star and double-star operators is overloaded and changes depending on the context they're used in, but I hope you get the idea.

---

<sup>3</sup>cf. “Fun With `*args` and `**kwargs`” chapter

Let's expand the technique laid out by the proxy decorator into a more useful practical example. Here's a trace decorator that logs function arguments and results during execution time:

```
def trace(func):
    def wrapper(*args, **kwargs):
        print(f'TRACE: calling {func.__name__}() '
              f'with {args}, {kwargs}')

        original_result = func(*args, **kwargs)

        print(f'TRACE: {func.__name__}() '
              f'returned {original_result!r}')

        return original_result
    return wrapper
```

Decorating a function with trace and then calling it will print the arguments passed to the decorated function and its return value. This is still somewhat of a “toy” example—but in a pinch it makes a great debugging aid:

```
@trace
def say(name, line):
    return f'{name}: {line}'

>>> say('Jane', 'Hello, World')
'TRACE: calling say() with ("Jane", "Hello, World"), {}'
'TRACE: say() returned "Jane: Hello, World"'
'Jane: Hello, World'
```

Speaking of debugging, there are some things you should keep in mind when debugging decorators:



## How to Write “Debuggable” Decorators

When you use a decorator, really what you’re doing is replacing one function with another. One downside of this process is that it “hides” some of the metadata attached to the original (undecorated) function.

For example, the original function name, its docstring, and parameter list are hidden by the wrapper closure:

```
def greet():  
    """Return a friendly greeting."""  
    return 'Hello!'  
  
decorated_greet = uppercase(greet)
```

If you try to access any of that function metadata, you’ll see the wrapper closure’s metadata instead:

```
>>> greet.__name__  
'greet'  
>>> greet.__doc__  
'Return a friendly greeting.'  
  
>>> decorated_greet.__name__  
'wrapper'  
>>> decorated_greet.__doc__  
None
```

This makes debugging and working with the Python interpreter awkward and challenging. Thankfully there’s a quick fix for this: the `functools.wraps` decorator included in Python’s standard library.<sup>4</sup>

You can use `functools.wraps` in your own decorators to copy over the lost metadata from the undecorated function to the decorator closure. Here’s an example:

---

<sup>4</sup>cf. Python Docs: “[functools.wraps](#)”

```
import functools

def uppercase(func):
    @functools.wraps(func)
    def wrapper():
        return func().upper()
    return wrapper
```

Applying `functools.wraps` to the wrapper closure returned by the decorator carries over the docstring and other metadata of the input function:

```
@uppercase
def greet():
    """Return a friendly greeting."""
    return 'Hello!'

>>> greet.__name__
'greet'
>>> greet.__doc__
'Return a friendly greeting.'
```

As a best practice, I'd recommend that you use `functools.wraps` in all of the decorators you write yourself. It doesn't take much time and it will save you (and others) debugging headaches down the road.

Oh, and congratulations—you've made it all the way to the end of this complicated chapter and learned a whole lot about decorators in Python. Great job!

## Key Takeaways

- Decorators define reusable building blocks you can apply to a callable to modify its behavior without permanently modifying the callable itself.

- The @ syntax is just a shorthand for calling the decorator on an input function. Multiple decorators on a single function are applied bottom to top (*decorator stacking*).
- As a debugging best practice, use the `functools.wraps` helper in your own decorators to carry over metadata from the undecorated callable to the decorated one.
- Just like any other tool in the software development toolbox, decorators are not a cure-all and they should not be overused. It's important to balance the need to “get stuff done” with the goal of “not getting tangled up in a horrible, unmaintainable mess of a code base.”

## 3.4 Fun With `*args` and `**kwargs`

I once pair-programmed with a smart Pythonista who would exclaim “argh!” and “kwargh!” every time he typed out a function definition with optional or keyword parameters. We got along great otherwise. I guess that’s what programming in academia does to people eventually.

Now, while easily mocked, `*args` and `**kwargs` parameters are nevertheless a highly useful feature in Python. And understanding their potency will make you a more effective developer.

So what are `*args` and `**kwargs` parameters used for? They allow a function to accept *optional* arguments, so you can create flexible APIs in your modules and classes:

```
def foo(required, *args, **kwargs):
    print(required)
    if args:
        print(args)
    if kwargs:
        print(kwargs)
```

The above function requires at least one argument called “required,” but it can accept extra positional and keyword arguments as well.

If we call the function with additional arguments, `args` will collect extra positional arguments as a tuple because the parameter name has a `*` prefix.

Likewise, `kwargs` will collect extra keyword arguments as a dictionary because the parameter name has a `**` prefix.

Both `args` and `kwargs` can be empty if no extra arguments are passed to the function.

As we call the function with various combinations of arguments, you’ll see how Python collects them inside the `args` and `kwargs` parameters

according to whether they’re positional or keyword arguments:

```
>>> foo()
TypeError:
"foo() missing 1 required positional arg: 'required'"

>>> foo('hello')
hello

>>> foo('hello', 1, 2, 3)
hello
(1, 2, 3)

>>> foo('hello', 1, 2, 3, key1='value', key2=999)
hello
(1, 2, 3)
{'key1': 'value', 'key2': 999}
```

I want to make it clear that calling the parameters `args` and `kwargs` is simply a naming convention. The previous example would work just as well if you called them `*parms` and `**argv`. The actual syntax is just the asterisk (\*) or double asterisk (\*\*), respectively.

However, I recommend that you stick with the accepted naming convention to avoid confusion. (And to get a chance to yell “argh!” and “kwargh!” every once in a while.)

## Forwarding Optional or Keyword Arguments

It’s possible to pass optional or keyword parameters from one function to another. You can do so by using the argument-unpacking operators \* and \*\* when calling the function you want to forward arguments to.<sup>5</sup>

---

<sup>5</sup>cf. “Function Argument Unpacking” chapter

This also gives you an opportunity to modify the arguments before you pass them along. Here's an example:

```
def foo(x, *args, **kwargs):
    kwargs['name'] = 'Alice'
    new_args = args + ('extra', )
    bar(x, *new_args, **kwargs)
```

This technique can be useful for subclassing and writing wrapper functions. For example, you can use it to extend the behavior of a parent class without having to replicate the full signature of its constructor in the child class. This can be quite convenient if you're working with an API that might change outside of your control:

```
class Car:
    def __init__(self, color, mileage):
        self.color = color
        self.mileage = mileage

class AlwaysBlueCar(Car):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.color = 'blue'

>>> AlwaysBlueCar('green', 48392).color
'blue'
```

The AlwaysBlueCar constructor simply passes on all arguments to its parent class and then overrides an internal attribute. This means if the parent class constructor changes, there's a good chance that AlwaysBlueCar would still function as intended.

The downside here is that the AlwaysBlueCar constructor now has a rather unhelpful signature—we don't know what arguments it expects without looking up the parent class.

Typically you wouldn't use this technique with your own class hierarchies. The more likely scenario would be that you'll want to modify or override behavior in some external class which you don't control.

But this is always dangerous territory, so best be careful (or you might soon have yet another reason to scream “argh!”).

One more scenario where this technique is potentially helpful is writing wrapper functions such as decorators. There you typically also want to accept arbitrary arguments to be passed through to the wrapped function.

And, if we can do it without having to copy and paste the original function's signature, that might be more maintainable:

```
def trace(f):
    @functools.wraps(f)
    def decorated_function(*args, **kwargs):
        print(f, args, kwargs)
        result = f(*args, **kwargs)
        print(result)
    return decorated_function

@trace
def greet(greeting, name):
    return '{} {}, {}'.format(greeting, name)

>>> greet('Hello', 'Bob')
<function greet at 0x1031c9158> ('Hello', 'Bob') {}
'Hello, Bob!'
```

With techniques like this one, it's sometimes difficult to balance the idea of making your code explicit enough and yet adhere to the *Don't Repeat Yourself (DRY)* principle. This will always be a tough choice to make. If you can get a second opinion from a colleague, I'd encourage you to ask for one.

## Key Takeaways

- `*args` and `**kwargs` let you write functions with a variable number of arguments in Python.
- `*args` collects extra positional arguments as a tuple. `**kwargs` collects the extra keyword arguments as a dictionary.
- The actual syntax is `*` and `**`. Calling them `args` and `kwargs` is just a convention (and one you should stick to).



## 3.5 Function Argument Unpacking

A really cool but slightly arcane feature is the ability to “unpack” function arguments from sequences and dictionaries with the `*` and `**` operators.

Let’s define a simple function to work with as an example:

```
def print_vector(x, y, z):  
    print('<%s, %s, %s>' % (x, y, z))
```

As you can see, this function takes three arguments (`x`, `y`, and `z`) and prints them in a nicely formatted way. We might use this function to pretty-print 3-dimensional vectors in our program:

```
>>> print_vector(0, 1, 0)  
<0, 1, 0>
```

Now depending on which data structure we choose to represent 3D vectors with, printing them with our `print_vector` function might feel a little awkward. For example, if our vectors are represented as tuples or lists we must explicitly specify the index for each component when printing them:

```
>>> tuple_vec = (1, 0, 1)  
>>> list_vec = [1, 0, 1]  
>>> print_vector(tuple_vec[0],  
                  tuple_vec[1],  
                  tuple_vec[2])  
<1, 0, 1>
```

Using a normal function call with separate arguments seems unnecessarily verbose and cumbersome. Wouldn’t it be much nicer if we could just “explode” a vector object into its three components and pass everything to the `print_vector` function all at once?

(Of course, you could simply redefine `print_vector` so that it takes a single parameter representing a vector object—but for the sake of having a simple example, we’ll ignore that option for now.)

Thankfully, there’s a better way to handle this situation in Python with *Function Argument Unpacking* using the `*` operator:

```
>>> print_vector(*tuple_vec)
<1, 0, 1>
>>> print_vector(*list_vec)
<1, 0, 1>
```

Putting a `*` before an iterable in a function call will *unpack* it and pass its elements as separate positional arguments to the called function.

This technique works for any iterable, including generator expressions. Using the `*` operator on a generator consumes all elements from the generator and passes them to the function:

```
>>> genexpr = (x * x for x in range(3))
>>> print_vector(*genexpr)
```

Besides the `*` operator for unpacking sequences like tuples, lists, and generators into positional arguments, there’s also the `**` operator for unpacking keyword arguments from dictionaries. Imagine our vector was represented as the following dict object:

```
>>> dict_vec = {'y': 0, 'z': 1, 'x': 1}
```

We could pass this dict to `print_vector` in much the same way using the `**` operator for unpacking:

```
>>> print_vector(**dict_vec)
<1, 0, 1>
```

Because dictionaries are unordered, this matches up dictionary values and function arguments based on the dictionary keys: the `x` argument receives the value associated with the `'x'` key in the dictionary.

If you were to use the single asterisk (`*`) operator to unpack the dictionary, keys would be passed to the function in random order instead:

```
>>> print_vector(*dict_vec)
<y, x, z>
```

Python's function argument unpacking feature gives you a lot of flexibility for free. Often this means you won't have to implement a class for a data type needed by your program. As a result, using simple built-in data structures like tuples or lists will suffice and help reduce the complexity of your code.

## Key Takeaways

- The `*` and `**` operators can be used to “unpack” function arguments from sequences and dictionaries.
- Using argument unpacking effectively can help you write more flexible interfaces for your modules and functions.

## 3.6 Nothing to Return Here

Python adds an implicit return `None` statement to the end of any function. Therefore, if a function doesn't specify a return value, it returns `None` by default.

This means you can replace `return None` statements with bare `return` statements or even leave them out completely and still get the same result:

```
def foo1(value):
    if value:
        return value
    else:
        return None

def foo2(value):
    """Bare return statement implies `return None`"""
    if value:
        return value
    else:
        return

def foo3(value):
    """Missing return statement implies `return None`"""
    if value:
        return value
```

All three functions properly return `None` if you pass them a falsy value as the sole argument:

```
>>> type(foo1(0))
<class 'NoneType'>

>>> type(foo2(0))
<class 'NoneType'>
```

```
>>> type(foo3(0))  
<class 'NoneType'>
```

Now, when is it a good idea to use this feature in your own Python code?

My rule of thumb is that if a function *doesn't have a return value* (other languages would call this a *procedure*), then I will leave out the return statement. Adding one would just be superfluous and confusing. An example for a procedure would be Python's built-in `print` function which is only called for its side-effects (printing text) and never for its return value.

Let's take a function like Python's built-in `sum`. It clearly has a logical return value, and typically `sum` wouldn't get called only for its side-effects. Its purpose is to add a sequence of numbers together and then deliver the result. Now, if a function *does* have a return value from a logical point of view, then you need to decide whether to use an implicit return or not.

On the one hand, you could argue that omitting an explicit return `None` statement makes the code more concise and therefore easier to read and understand. Subjectively, you might also say it makes the code “prettier.”

On the other hand, it might surprise some programmers that Python behaves this way. When it comes to writing clean and maintainable code, surprising behavior is rarely a good sign.

For example, I've been using an “implicit return statement” in one of the code samples in an earlier revision of the book. I didn't mention what I was doing—I just wanted a nice short code sample to explain some other feature in Python.

Eventually I started getting a steady stream of emails pointing me to “the missing return statement” in that code example. Python's implicit return behavior was clearly *not* obvious to everybody and was a dis-

traction in this case. I added a note to make it clear what was going on, and the emails stopped.

Don't get me wrong—I love writing clean and “beautiful” code as much as anyone. And I also used to feel strongly that programmers should know the ins and outs of the language they're working with.

But when you consider the maintenance impact of even such a simple misunderstanding, it might make sense to lean towards writing more explicit and clear code. After all, *code is communication*.

## Key Takeaways

- If a function doesn't specify a return value, it returns `None`. Whether to explicitly return `None` is a stylistic decision.
- This is a core Python feature but your code might communicate its intent more clearly with an explicit `return None` statement.

# **Chapter 4**

## **Classes & OOP**

## 4.1 Object Comparisons: “is” vs “==”

When I was a kid, our neighbors had two twin cats. They looked seemingly identical—the same charcoal fur and the same piercing green eyes. Some personality quirks aside, you couldn’t tell them apart just from looking at them. But of course, they were two different cats, two separate beings, even though they looked exactly the same.

That brings me to the difference in meaning between *equal* and *identical*. And this difference is crucial to understanding how Python’s `is` and `==` comparison operators behave.

The `==` operator compares by checking for *equality*: if these cats were Python objects and we compared them with the `==` operator, we’d get “both cats are equal” as an answer.

The `is` operator, however, compares *identities*: if we compared our cats with the `is` operator, we’d get “these are two different cats” as an answer.

But before I get all tangled up in this ball-of-twine cat analogy, let’s take a look at some real Python code.

First, we’ll create a new list object and name it `a`, and then define another variable (`b`) that points to the same list object:

```
>>> a = [1, 2, 3]
>>> b = a
```

Let’s inspect these two variables. We can see that they point to identical-looking lists:

```
>>> a
[1, 2, 3]
>>> b
[1, 2, 3]
```



Because the two list objects look the same, we’ll get the expected result when we compare them for equality by using the == operator:

```
>>> a == b
True
```

However, that doesn’t tell us whether a and b are actually pointing to the same object. Of course, we know they are because we assigned them earlier, but suppose we didn’t know—how might we find out?

The answer is to compare both variables with the is operator. This confirms that both variables are in fact pointing to one list object:

```
>>> a is b
True
```

Let’s see what happens when we create an identical copy of our list object. We can do that by calling list() on the existing list to create a copy we’ll name c:

```
>>> c = list(a)
```

Again you’ll see that the new list we just created looks identical to the list object pointed to by a and b:

```
>>> c
[1, 2, 3]
```

Now this is where it gets interesting. Let’s compare our list copy c with the initial list a using the == operator. What answer do you expect to see?

```
>>> a == c
True
```

Okay, I hope this was what you expected. What this result tells us is that `c` and `a` have the same contents. They’re considered equal by Python. But are they actually pointing to the same object? Let’s find out with the `is` operator:

```
>>> a is c
False
```

Boom! This is where we get a different result. Python is telling us that `c` and `a` are pointing to two different objects, even though their contents might be the same.

So, to recap, let’s try and break down the difference between `is` and `==` into two short definitions:

- An `is` expression evaluates to `True` if two variables point to the same (identical) object.
- An `==` expression evaluates to `True` if the objects referred to by the variables are equal (have the same contents).

Just remember to think of twin cats (dogs should work, too) whenever you need to decide between using `is` and `==` in Python. If you do that, you’ll be fine.

## 4.2 String Conversion (Every Class Needs a `__repr__`)

When you define a custom class in Python and then try to print one of its instances to the console (or inspect it in an interpreter session), you get a relatively unsatisfying result. The default “to string” conversion behavior is basic and lacks detail:

```
class Car:
    def __init__(self, color, mileage):
        self.color = color
        self.mileage = mileage

>>> my_car = Car('red', 37281)
>>> print(my_car)
<__console__.Car object at 0x109b73da0>
>>> my_car
<__console__.Car object at 0x109b73da0>
```

By default all you get is a string containing the class name and the id of the object instance (which is the object’s memory address in CPython.) That’s better than *nothing*, but it’s also not very useful.

You might find yourself trying to work around this by printing attributes of the class directly, or even by adding a custom `to_string()` method to your classes:

```
>>> print(my_car.color, my_car.mileage)
red 37281
```

The general idea here is the right one—but it ignores the conventions and built-in mechanisms Python uses to handle how objects are represented as strings.

Instead of building your own to-string conversion machinery, you’ll be better off adding the `__str__` and `__repr__` “dunder” methods to

your class. They are the Pythonic way to control how objects are converted to strings in different situations.<sup>1</sup>

Let's take a look at how these methods work in practice. To get started, we're going to add a `__str__` method to the `Car` class we defined earlier:

```
class Car:
    def __init__(self, color, mileage):
        self.color = color
        self.mileage = mileage

    def __str__(self):
        return f'a {self.color} car'
```

When you try printing or inspecting a `Car` instance now, you'll get a different, slightly improved result:

```
>>> my_car = Car('red', 37281)
>>> print(my_car)
'a red car'
>>> my_car
<__console__.Car object at 0x109ca24e0>
```

Inspecting the car object in the console still gives us the previous result containing the object's id. But *printing* the object resulted in the string returned by the `__str__` method we added.

`__str__` is one of Python's “dunder” (double-underscore) methods and gets called when you try to convert an object into a string through the various means that are available:

```
>>> print(my_car)
a red car
```

---

<sup>1</sup>cf. Python Docs: “The Python Data Model”

```
>>> str(my_car)
'a red car'
>>> '{}'.format(my_car)
'a red car'
```

With a proper `__str__` implementation, you won't have to worry about printing object attributes directly or writing a separate `to_string()` function. It's the Pythonic way to control string conversion.

By the way, some people refer to Python's “dunder” methods as “magic methods.” But these methods are not supposed to be *magical* in any way. The fact that these methods start and end in double underscores is simply a naming convention to flag them as core Python features. It also helps avoid naming collisions with your own methods and attributes. The object constructor `__init__` follows the same convention, and there's nothing magical or arcane about it.

Don't be afraid to use Python's dunder methods—they're meant to help you.

### `__str__` vs `__repr__`

Now, our string conversion story doesn't end there. Did you see how inspecting `my_car` in an interpreter session still gave that odd `<Car object at 0x109ca24e0>` result?

This happened because there are actually *two* dunder methods that control how objects are converted to strings in Python 3. The first one is `__str__`, and you just learned about it. The second one is `__repr__`, and the way it works is similar to `__str__`, but it is used in different situations. (Python 2.x also has a `__unicode__` method that I'll touch on a little later.)

Here's a simple experiment you can use to get a feel for when `__str__` or `__repr__` is used. Let's redefine our car class so it contains both *to-string* dunder methods with outputs that are easy to distinguish:

```
class Car:
    def __init__(self, color, mileage):
        self.color = color
        self.mileage = mileage

    def __repr__(self):
        return '__repr__ for Car'

    def __str__(self):
        return '__str__ for Car'
```

Now, when you play through the previous examples you can see which method controls the string conversion result in each case:

```
>>> my_car = Car('red', 37281)
>>> print(my_car)
__str__ for Car
>>> '{}'.format(my_car)
'__str__ for Car'
>>> my_car
__repr__ for Car
```

This experiment confirms that inspecting an object in a Python interpreter session simply prints the result of the object's `__repr__`.

Interestingly, containers like lists and dicts always use the result of `__repr__` to represent the objects they contain. Even if you call `str` on the container itself:

```
str([my_car])
'[__repr__ for Car]'
```

To manually choose between both string conversion methods, for example, to express your code's intent more clearly, it's best to use the built-in `str()` and `repr()` functions. Using them is preferable over

calling the object's `__str__` or `__repr__` directly, as it looks nicer and gives the same result:

```
>>> str(my_car)
'__str__ for Car'
>>> repr(my_car)
'__repr__ for Car'
```

Even with this investigation complete, you might be wondering what the “real-world” difference is between `__str__` and `__repr__`. They both seem to serve the same purpose, so it might be unclear when to use each.

With questions like that, it's usually a good idea to look into what the Python standard library does. Time to devise another experiment. We'll create a `datetime.date` object and find out how it uses `__repr__` and `__str__` to control string conversion:

```
>>> import datetime
>>> today = datetime.date.today()
```

The result of the date object's `__str__` function should primarily be *readable*. It's meant to return a concise textual representation for human consumption—something you'd feel comfortable displaying to a user. Therefore, we get something that looks like an ISO date format when we call `str()` on the date object:

```
>>> str(today)
'2017-02-02'
```

With `__repr__`, the idea is that its result should be, above all, *unambiguous*. The resulting string is intended more as a debugging aid for developers. And for that it needs to be as explicit as possible about what this object is. That's why you'll get a more elaborate result calling `repr()` on the object. It even includes the full module and class name:

```
>>> repr(today)
'datetime.date(2017, 2, 2)'
```

We could copy and paste the string returned by `__repr__` and execute it as valid Python to recreate the original date object. This is a neat approach and a good goal to keep in mind while writing your own `reprs`.

On the other hand, I find that it is quite difficult to put into practice. Usually it won't be worth the trouble and it'll just create extra work for you. My rule of thumb is to make my `__repr__` strings unambiguous and helpful for developers, but I don't expect them to be able to restore an object's complete state.

### Why Every Class Needs a `__repr__`

If you don't add a `__str__` method, Python falls back on the result of `__repr__` when looking for `__str__`. Therefore, I recommend that you always add at least a `__repr__` method to your classes. This will guarantee a useful string conversion result in almost all cases, with a minimum of implementation work.

Here's how to add basic string conversion support to your classes quickly and efficiently. For our `Car` class we might start with the following `__repr__`:

```
def __repr__(self):
    return f'Car({self.color!r}, {self.mileage!r})'
```

Please note that I'm using the `!r` conversion flag to make sure the output string uses `repr(self.color)` and `repr(self.mileage)` instead of `str(self.color)` and `str(self.mileage)`.

This works nicely, but one downside is that we've repeated the class name inside the format string. A trick you can use here to avoid this



repetition is to use the object's `__class__.__name__` attribute, which will always reflect the class' name as a string.

The benefit is you won't have to modify the `__repr__` implementation when the class name changes. This makes it easy to adhere to the *Don't Repeat Yourself (DRY)* principle:

```
def __repr__(self):  
    return (f'{self.__class__.__name__}(''  
            f'{self.color!r}, {self.mileage!r})''')
```

The downside of this implementation is that the format string is quite long and unwieldy. But with careful formatting, you can keep the code nice and PEP 8 compliant.

With the above `__repr__` implementation, we get a useful result when we inspect the object or call `repr()` on it directly:

```
>>> repr(my_car)  
'Car(red, 37281)'
```

Printing the object or calling `str()` on it returns the same string because the default `__str__` implementation simply calls `__repr__`:

```
>>> print(my_car)  
'Car(red, 37281)'  
>>> str(my_car)  
'Car(red, 37281)'
```

I believe this approach provides the most value with a modest amount of implementation work. It's also a fairly cookie-cutter approach that can be applied without much deliberation. For this reason, I always try to add a basic `__repr__` implementation to my classes.

Here's a complete example for Python 3, including an optional `__str__` implementation:

```
class Car:
    def __init__(self, color, mileage):
        self.color = color
        self.mileage = mileage

    def __repr__(self):
        return (f'{self.__class__.__name__}('
                f'{self.color!r}, {self.mileage!r})')

    def __str__(self):
        return f'a {self.color} car'
```

## Python 2.x Differences: `__unicode__`

In Python 3 there's one data type to represent text across the board: `str`. It holds unicode characters and can represent most of the world's writing systems.

Python 2.x uses a different data model for strings.<sup>2</sup> There are two types to represent text: `str`, which is limited to the ASCII character set, and `unicode`, which is equivalent to Python 3's `str`.

Due to this difference, there's yet another dunder method in the mix for controlling string conversion in Python 2: `__unicode__`. In Python 2, `__str__` returns *bytes*, whereas `__unicode__` returns *characters*.

For most intents and purposes, `__unicode__` is the newer and preferred method to control string conversion. There's also a built-in `unicode()` function to go along with it. It calls the respective dunder method, similar to how `str()` and `repr()` work.

So far so good. Now, it gets a little more quirky when you look at the rules for when `__str__` and `__unicode__` are called in Python 2:

The `print` statement and `str()` call `__str__`. The `unicode()` built-in

---

<sup>2</sup>cf. Python 2 Docs: "Data Model"

calls `__unicode__` if it exists, and otherwise falls back to `__str__` and decodes the result with the system text encoding.

Compared to Python 3, these special cases complicate the text conversion rules somewhat. But there is a way to simplify things again for practical purposes. Unicode is the preferred and future-proof way of handling text in your Python programs.

So generally, what I would recommend you do in Python 2.x is to put all of your string formatting code inside the `__unicode__` method and then create a stub `__str__` implementation that returns the unicode representation encoded as UTF-8:

```
def __str__(self):  
    return unicode(self).encode('utf-8')
```

The `__str__` stub will be the same for most classes you write, so you can just copy and paste it around as needed (or put it into a base class where it makes sense). All of your string conversion code that is meant for non-developer use then lives in `__unicode__`.

Here's a complete example for Python 2.x:

```
class Car(object):  
    def __init__(self, color, mileage):  
        self.color = color  
        self.mileage = mileage  
  
    def __repr__(self):  
        return '{}({!r}, {!r})'.format(  
            self.__class__.__name__,  
            self.color, self.mileage)  
  
    def __unicode__(self):  
        return u'a {self.color} car'.format(  
            self=self)
```

```
def __str__(self):  
    return unicode(self).encode('utf-8')
```

### Key Takeaways

- You can control to-string conversion in your own classes using the `__str__` and `__repr__` “dunder” methods.
- The result of `__str__` should be readable. The result of `__repr__` should be unambiguous.
- Always add a `__repr__` to your classes. The default implementation for `__str__` just calls `__repr__`.
- Use `__unicode__` instead of `__str__` in Python 2.

## 4.3 Defining Your Own Exception Classes

When I started using Python, I was hesitant to write custom exception classes in my code. But defining your own error types can be of great value. You'll make potential error cases stand out clearly, and as a result, your functions and modules will become more maintainable. You can also use custom error types to provide additional debugging information.

All of this will improve your Python code and make it easier to understand, easier to debug, and more maintainable. Defining your own exception classes is not that hard when you break it down to a few simple examples. In this chapter I'll walk you through the main points you need to remember.

Let's say you wanted to validate an input string representing a person's name in your application. A toy example for a name validator function might look like this:

```
def validate(name):  
    if len(name) < 10:  
        raise ValueError
```

If the validation fails, it throws a `ValueError` exception. That seems fitting and kind of Pythonic already. So far, so good.

However, there's a downside to using a "high-level" generic exception class like `ValueError`. Imagine one of your teammates calls this function as part of a library and doesn't know much about its internals. When a name fails to validate, it'll look like this in the debug stack trace:

```
>>> validate('joe')  
Traceback (most recent call last):  
  File "<input>", line 1, in <module>
```

```
validate('joe')
File "<input>", line 3, in validate
    raise ValueError
ValueError
```

This stack trace isn't really all that helpful. Sure, we know that something went wrong and that the problem had to do with an "incorrect value" of sorts, but to be able to fix the problem your teammate almost certainly has to look up the implementation of `validate()`. However, reading code costs time. And it can add up quickly.

Luckily we can do better. Let's introduce a custom exception type to represent a failed name validation. We'll base our new exception class on Python's built-in `ValueError`, but make it speak for itself by giving it a more explicit name:

```
class NameTooShortError(ValueError):
    pass

def validate(name):
    if len(name) < 10:
        raise NameTooShortError(name)
```

Now we have a "self-documenting" `NameTooShortError` exception type that extends the built-in `ValueError` class. Generally, you'll want to either derive your custom exceptions from the root `Exception` class or the other built-in Python exceptions like `ValueError` or `TypeError`—whichever feels appropriate.

Also, see how we're now passing the `name` variable to the constructor of our custom exception class when we instantiate it inside `validate`? The new implementation results in a much nicer stack trace for your colleague:

```
>>> validate('jane')
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    validate('jane')
  File "<input>", line 3, in validate
    raise NameTooShortError(name)
NameTooShortError: jane
```

Once again, try to put yourself in your teammate's shoes. Custom exception classes make it much easier to understand what's going on when things go wrong (and eventually they always do).

The same is true even if you're working on a code base all by yourself. A few weeks or months down the road you'll have a much easier time maintaining your code if it's well-structured.

By spending just 30 seconds on defining a simple exception class, this code snippet became much more communicative already. But let's keep going. There's more to cover.

Whenever you're publicly releasing a Python package, or even if you're creating a reusable module for your company, it's good practice to create a custom exception base class for the module and then derive all of your other exceptions from it.

Here's how to create a custom exception hierarchy for all exceptions in a module or package. The first step is to declare a base class that all of our concrete errors will inherit from:

```
class BaseValidationError(ValueError):
    pass
```

Now, all of our "real" error classes can be derived from the base error class. This gives a nice and clean exception hierarchy with little extra effort:

```
class NameTooShortError(BaseValidationError):  
    pass  
  
class NameTooLongError(BaseValidationError):  
    pass  
  
class NameTooCuteError(BaseValidationError):  
    pass
```

For example, this allows users of your package to write *try...except* statements that can handle all of the errors from this package without having to catch them manually:

```
try:  
    validate(name)  
except BaseValidationError as err:  
    handle_validation_error(err)
```

People can still catch more specific exceptions that way, but if they don't want to, at least they won't have to resort to snapping up all exceptions with a catchall *except* statement. This is generally considered an anti-pattern—it can silently swallow and hide unrelated errors and make your programs much harder to debug.

Of course you can take this idea further and logically group your exceptions into fine grained sub-hierarchies. But be careful—it's easy to introduce unnecessary complexity by going overboard with this.

In conclusion, defining custom exception classes makes it easier for your users to adopt an *it's easier to ask for forgiveness than permission* (EAFP) coding style that's considered more Pythonic.

## Key Takeaways

- Defining your own exception types will state your code's intent more clearly and make it easier to debug.



- Derive your custom exceptions from Python's built-in Exception class or from more specific exception classes like ValueError or KeyError.
- You can use inheritance to define logically grouped exception hierarchies.

## 4.4 Cloning Objects for Fun and Profit

Assignment statements in Python do not create copies of objects, they only bind names to an object. For immutable objects, that usually doesn't make a difference.

But for working with mutable objects or collections of mutable objects, you might be looking for a way to create “real copies” or “clones” of these objects.

Essentially, you'll sometimes want copies that you can modify *without* automatically modifying the original at the same time. In this chapter I'm going to give you the rundown on how to copy or “clone” objects in Python and some of the caveats involved.

Let's start by looking at how to copy Python's built-in collections. Python's built-in mutable collections like lists, dicts, and sets can be copied by calling their factory functions on an existing collection:

```
new_list = list(original_list)
new_dict = dict(original_dict)
new_set = set(original_set)
```

However, this method won't work for custom objects and, on top of that, it only creates *shallow copies*. For compound objects like lists, dicts, and sets, there's an important difference between *shallow* and *deep* copying:

A *shallow copy* means constructing a new collection object and then populating it with references to the child objects found in the original. In essence, a shallow copy is only *one level deep*. The copying process does not recurse and therefore won't create copies of the child objects themselves.

A *deep copy* makes the copying process recursive. It means first constructing a new collection object and then recursively populating it with copies of the child objects found in the original. Copying an ob-

ject this way walks the whole object tree to create a fully independent clone of the original object and all of its children.

I know, that was a bit of a mouthful. So let's look at some examples to drive home this difference between deep and shallow copies.

## Making Shallow Copies

In the example below, we'll create a new nested list and then *shallowly* copy it with the `list()` factory function:

```
>>> xs = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> ys = list(xs) # Make a shallow copy
```

This means `ys` will now be a new and independent object with the same contents as `xs`. You can verify this by inspecting both objects:

```
>>> xs
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> ys
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

To confirm `ys` really is independent from the original, let's devise a little experiment. You could try and add a new sublist to the original (`xs`) and then check to make sure this modification didn't affect the copy (`ys`):

```
>>> xs.append(['new sublist'])
>>> xs
[[1, 2, 3], [4, 5, 6], [7, 8, 9], ['new sublist']]
>>> ys
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

As you can see, this had the expected effect. Modifying the copied list at a “superficial” level was no problem at all.

However, because we only created a *shallow* copy of the original list, `ys` still contains references to the original child objects stored in `xs`.

These children were *not* copied. They were merely referenced again in the copied list.

Therefore, when you modify one of the child objects in `xs`, this modification will be reflected in `ys` as well—that's because *both lists share the same child objects*. The copy is only a shallow, one level deep copy:

```
>>> xs[1][0] = 'X'
>>> xs
[[1, 2, 3], ['X', 5, 6], [7, 8, 9], ['new sublist']]
>>> ys
[[1, 2, 3], ['X', 5, 6], [7, 8, 9]]
```

In the above example we (seemingly) only made a change to `xs`. But it turns out that *both* sublists at index 1 in `xs` *and* `ys` were modified. Again, this happened because we had only created a *shallow* copy of the original list.

Had we created a *deep* copy of `xs` in the first step, both objects would've been fully independent. This is the practical difference between shallow and deep copies of objects.

Now you know how to create shallow copies of some of the built-in collection classes, and you know the difference between shallow and deep copying. The questions we still want answers for are:

- How can you create deep copies of built-in collections?
- How can you create copies (shallow and deep) of arbitrary objects, including custom classes?

The answer to these questions lies in the `copy` module in the Python standard library. This module provides a simple interface for creating shallow and deep copies of arbitrary Python objects.

## Making Deep Copies

Let's repeat the previous list-copying example, but with one important difference. This time we're going to create a *deep* copy using the `deepcopy()` function defined in the `copy` module instead:

```
>>> import copy
>>> xs = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> zs = copy.deepcopy(xs)
```

When you inspect `xs` and its clone `zs` that we created with `copy.deepcopy()`, you'll see that they both look identical again—just like in the previous example:

```
>>> xs
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> zs
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

However, if you make a modification to one of the child objects in the original object (`xs`), you'll see that this modification won't affect the deep copy (`zs`).

Both objects, the original and the copy, are fully independent this time. `xs` was cloned recursively, including all of its child objects:

```
>>> xs[1][0] = 'X'
>>> xs
[[1, 2, 3], ['X', 5, 6], [7, 8, 9]]
>>> zs
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

You might want to take some time to sit down with the Python interpreter and play through these examples right about now. Wrapping your head around copying objects is easier when you get to experience and play with the examples firsthand.

By the way, you can also create shallow copies using a function in the `copy` module. The `copy.copy()` function creates shallow copies of objects.

This is useful if you need to clearly communicate that you're creating a shallow copy somewhere in your code. Using `copy.copy()` lets you indicate this fact. However, for built-in collections it's considered more Pythonic to simply use the list, dict, and set factory functions to create shallow copies.

## Copying Arbitrary Objects

The question we still need to answer is how do we create copies (shallow and deep) of arbitrary objects, including custom classes. Let's take a look at that now.

Again the `copy` module comes to our rescue. Its `copy.copy()` and `copy.deepcopy()` functions can be used to duplicate any object.

Once again, the best way to understand how to use these is with a simple experiment. I'm going to base this on the previous list-copying example. Let's start by defining a simple 2D point class:

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f'Point({self.x!r}, {self.y!r})'
```

I hope you agree that this was pretty straightforward. I added a `__repr__()` implementation so that we can easily inspect objects created from this class in the Python interpreter.

Next up, we'll create a `Point` instance and then (shallowly) copy it, using the `copy` module:

```
>>> a = Point(23, 42)
>>> b = copy.copy(a)
```

If we inspect the contents of the original `Point` object and its (shallow) clone, we see what we'd expect:

```
>>> a
Point(23, 42)
>>> b
Point(23, 42)
>>> a is b
False
```

Here's something else to keep in mind. Because our point object uses primitive types (ints) for its coordinates, there's no difference between a shallow and a deep copy in this case. But I'll expand the example in a second.

Let's move on to a more complex example. I'm going to define another class to represent 2D rectangles. I'll do it in a way that allows us to create a more complex object hierarchy—my rectangles will use `Point` objects to represent their coordinates:

```
class Rectangle:
    def __init__(self, topleft, bottomright):
        self.topleft = topleft
        self.bottomright = bottomright

    def __repr__(self):
        return (f'Rectangle({self.topleft!r}, '
                f'{self.bottomright!r})')
```

Again, first we're going to attempt to create a shallow copy of a rectangle instance:

```
rect = Rectangle(Point(0, 1), Point(5, 6))
srect = copy.copy(rect)
```

If you inspect the original rectangle and its copy, you'll see how nicely the `__repr__()` override is working out, and that the shallow copy process worked as expected:

```
>>> rect
Rectangle(Point(0, 1), Point(5, 6))
>>> srect
Rectangle(Point(0, 1), Point(5, 6))
>>> rect is srect
False
```

Remember how the previous list example illustrated the difference between deep and shallow copies? I'm going to use the same approach here. I'll modify an object deeper in the object hierarchy, and then you'll see this change reflected in the (shallow) copy as well:

```
>>> rect.topleft.x = 999
>>> rect
Rectangle(Point(999, 1), Point(5, 6))
>>> srect
Rectangle(Point(999, 1), Point(5, 6))
```

I hope this behaved how you expected it to. Next, I'll create a deep copy of the original rectangle. Then I'll apply another modification and you'll see which objects are affected:

```
>>> drect = copy.deepcopy(srect)
>>> drect.topleft.x = 222
>>> drect
Rectangle(Point(222, 1), Point(5, 6))
>>> rect
```



```
Rectangle(Point(999, 1), Point(5, 6))
>>> srect
Rectangle(Point(999, 1), Point(5, 6))
```

Voila! This time the deep copy (`direct`) is fully independent of the original (`rect`) and the shallow copy (`srect`).

We’ve covered a lot of ground here, and there are still some finer points to copying objects.

It pays to go deep (ha!) on this topic, so you may want to study up on the `copy` module documentation.<sup>3</sup> For example, objects can control how they’re copied by defining the special methods `__copy__()` and `__deepcopy__()` on them. Have fun!

### Key Takeaways

- Making a shallow copy of an object won’t clone child objects. Therefore, the copy is not fully independent of the original.
- A deep copy of an object will recursively clone child objects. The clone is fully independent of the original, but creating a deep copy is slower.
- You can copy arbitrary objects (including custom classes) with the `copy` module.

---

<sup>3</sup>cf. [Python docs: “Shallow and deep copy operations”](#)

## 4.5 Abstract Base Classes Keep Inheritance in Check

Abstract Base Classes (ABCs) ensure that derived classes implement particular methods from the base class. In this chapter you'll learn about the benefits of abstract base classes and how to define them with Python's built-in `abc` module.

So what are Abstract Base Classes good for? A while ago I had a discussion at work about which pattern to use for implementing a maintainable class hierarchy in Python. More specifically, the goal was to define a simple class hierarchy for a service backend in the most programmer-friendly and maintainable way.

We had a `BaseService` class that defined a common interface and several concrete implementations. The concrete implementations do different things but all of them provide the same interface (`MockService`, `RealService`, and so on). To make this relationship explicit, the concrete implementations all subclass `BaseService`.

To make this code as maintainable and programmer-friendly as possible we wanted to make sure that:

- instantiating the base class is impossible; and
- forgetting to implement interface methods in one of the subclasses raises an error as early as possible.

Now why would you want to use Python's `abc` module to solve this problem? The above design is pretty common in more complex systems. To enforce that a derived class implements a number of methods from the base class, something like this Python idiom is typically used:

```
class Base:
    def foo(self):
        raise NotImplementedError()
```

```
def bar(self):  
    raise NotImplementedError()  
  
class Concrete(Base):  
    def foo(self):  
        return 'foo() called'  
  
    # Oh no, we forgot to override bar()...  
    # def bar(self):  
    #     return "bar() called"
```

So, what do we get from this first attempt at solving the problem? Calling methods on an instance of `Base` correctly raises `NotImplementedError` exceptions:

```
>>> b = Base()  
>>> b.foo()  
NotImplementedError
```

Furthermore, instantiating and using `Concrete` works as expected. And, if we call an unimplemented method like `bar()` on it, this also raises an exception:

```
>>> c = Concrete()  
>>> c.foo()  
'foo() called'  
>>> c.bar()  
NotImplementedError
```

This first implementation is decent, but it isn't perfect yet. The downsides here are that we can still:

- instantiate `Base` just fine without getting an error; and

- provide incomplete subclasses—instantiating `Concrete` will not raise an error until we call the missing method `bar()`.

With Python’s `abc` module that was added in Python 2.6,<sup>4</sup> we can do better and solve these remaining issues. Here’s an updated implementation using an Abstract Base Class defined with the `abc` module:

```
from abc import ABCMeta, abstractmethod

class Base(metaclass=ABCMeta):
    @abstractmethod
    def foo(self):
        pass

    @abstractmethod
    def bar(self):
        pass

class Concrete(Base):
    def foo(self):
        pass

    # We forget to declare bar() again...
```

This still behaves as expected and creates the correct class hierarchy:

```
assert isinstance(Concrete, Base)
```

Yet, we do get another very useful benefit here. Subclasses of `Base` raise a `TypeError` *at instantiation time* whenever we forget to implement any abstract methods. The raised exception tells us which method or methods we’re missing:

---

<sup>4</sup>cf. [Python Docs: abc module](#)

```
>>> c = Concrete()
TypeError:
"Can't instantiate abstract class Concrete
with abstract methods bar"
```

Without `abc`, we'd only get a `NotImplementedError` if a missing method was actually called. Being notified about missing methods at instantiation time is a great advantage. It makes it more difficult to write invalid subclasses. This might not be a big deal if you're writing new code, but a few weeks or months down the line, I promise it'll be helpful.

This pattern is not a full replacement for compile-time type checking, of course. However, I found it often makes my class hierarchies more robust and more readily maintainable. Using ABCs states the programmer's intent clearly and thus makes the code more communicative. I'd encourage you to read the `abc` module documentation and to keep an eye out for situations where applying this pattern makes sense.

### Key Takeaways

- Abstract Base Classes (ABCs) ensure that derived classes implement particular methods from the base class at instantiation time.
- Using ABCs can help avoid bugs and make class hierarchies easier to maintain.

## 4.6 What Namedtuples Are Good For

Python comes with a specialized “namedtuple” container type that doesn’t seem to get the attention it deserves. It’s one of those amazing features in Python that’s hidden in plain sight.

Namedtuples can be a great alternative to defining a class manually, and they have some other interesting features that I want to introduce you to in this chapter.

Now, what’s a namedtuple and what makes it so special? A good way to think about namedtuples is to view them as an extension of the built-in tuple data type.

Python’s tuples are a simple data structure for grouping arbitrary objects. Tuples are also immutable—they cannot be modified once they’ve been created. Here’s a brief example:

```
>>> tup = ('hello', object(), 42)
>>> tup
('hello', <object object at 0x105e76b70>, 42)
>>> tup[2]
42
>>> tup[2] = 23
TypeError:
"'tuple' object does not support item assignment"
```

One downside of plain tuples is that the data you store in them can only be pulled out by accessing it through integer indexes. You can’t give names to individual properties stored in a tuple. This can impact code readability.

Also, a tuple is always an ad-hoc structure. It’s hard to ensure that two tuples have the same number of fields and the same properties stored on them. This makes it easy to introduce “slip-of-the-mind” bugs by mixing up the field order.

## Namedtuples to the Rescue

Namedtuples aim to solve these two problems.

First of all, namedtuples are immutable containers, just like regular tuples. Once you store data in top-level attribute on a namedtuple, you can't modify it by updating the attribute. All attributes on a namedtuple object follow the “write once, read many” principle.

Besides that, namedtuples are, well...*named tuples*. Each object stored in them can be accessed through a unique (human-readable) identifier. This frees you from having to remember integer indexes, or resorting to workarounds like defining integer constants as mnemonics for your indexes.

Here's what a namedtuple looks like:

```
>>> from collections import namedtuple
>>> Car = namedtuple('Car' , 'color mileage')
```

Namedtuples were added to the standard library in Python 2.6. To use them, you need to import the collections module. In the above example, I defined a simple Car data type with two fields: color and mileage.

You might be wondering why I'm passing the string 'Car' as the first argument to the namedtuple factory function in this example.

This parameter is referred to as the “typename” in the Python docs. It's the name of the new class that's being created by calling the namedtuple function.

Since namedtuple has no way of knowing what the name of the variable is we're assigning the resulting class to, we need to explicitly tell it which class name we want to use. The class name is used in the docstring and the `__repr__` implementation that namedtuple automatically generates for us.

And there's another syntactic oddity in this example—why are we passing the fields as a string that encodes their names as 'color mileage'?

The answer is that `namedtuple`'s factory function calls `split()` on the field names string to parse it into a list of field names. So this is really just a shorthand for the following two steps:

```
>>> 'color mileage'.split()
['color', 'mileage']
>>> Car = namedtuple('Car', ['color', 'mileage'])
```

Of course, you can also pass in a list with string field names directly if you prefer how that looks. The advantage of using a proper list is that it's easier to reformat this code if you need to split it across multiple lines:

```
>>> Car = namedtuple('Car', [
...     'color',
...     'mileage',
... ])
```

Whatever you decide, you can now create new “car” objects with the `Car` factory function. It behaves as if you had defined a `Car` class manually and given it a constructor accepting a “color” and a “mileage” value:

```
>>> my_car = Car('red', 3812.4)
>>> my_car.color
'red'
>>> my_car.mileage
3812.4
```

Besides accessing the values stored in a `namedtuple` by their identifiers, you can still access them by their index. That way, `namedtuples` can be used as a drop-in replacement for regular tuples:



```
>>> my_car[0]
'red'
>>> tuple(my_car)
('red', 3812.4)
```

Tuple unpacking and the `*`-operator for function argument unpacking also work as expected:

```
>>> color, mileage = my_car
>>> print(color, mileage)
red 3812.4
>>> print(*my_car)
red 3812.4
```

You'll even get a nice string representation for your namedtuple object for free, which saves some typing and verbosity:

```
>>> my_car
Car(color='red' , mileage=3812.4)
```

Like tuples, namedtuples are immutable. When you try to overwrite one of their fields, you'll get an `AttributeError` exception:

```
>>> my_car.color = 'blue'
AttributeError: "can't set attribute"
```

Namedtuple objects are implemented as regular Python classes internally. When it comes to memory usage, they are also “better” than regular classes and just as memory efficient as regular tuples.

A good way to view them is to think that *namedtuples are a memory-efficient shortcut to defining an immutable class in Python manually.*

## Subclassing Namedtuples

Since they are built on top of regular Python classes, you can even add methods to a namedtuple object. For example, you can extend a namedtuple's class like any other class and add methods and new properties to it that way. Here's an example:

```
Car = namedtuple('Car', 'color mileage')

class MyCarWithMethods(Car):
    def hexcolor(self):
        if self.color == 'red':
            return '#ff0000'
        else:
            return '#000000'
```

We can now create `MyCarWithMethods` objects and call their `hexcolor()` method, just as expected:

```
>>> c = MyCarWithMethods('red', 1234)
>>> c.hexcolor()
'#ff0000'
```

However, this might be a little clunky. It might be worth doing if you want a class with immutable properties, but it's also easy to shoot yourself in the foot here.

For example, adding a new *immutable* field is tricky because of how namedtuples are structured internally. The easiest way to create hierarchies of namedtuples is to use the base tuple's `_fields` property:

```
>>> Car = namedtuple('Car', 'color mileage')
>>> ElectricCar = namedtuple(
...     'ElectricCar', Car._fields + ('charge',))
```

This gives the desired result:

```
>>> ElectricCar('red', 1234, 45.0)
ElectricCar(color='red', mileage=1234, charge=45.0)
```

## Built-in Helper Methods

Besides the `_fields` property, each `namedtuple` instance also provides a few more helper methods you might find useful. Their names all start with a single underscore character (`_`) which usually signals that a method or property is “private” and not part of the stable public interface of a class or module.

With `namedtuples`, the underscore naming convention has a different meaning though. These helper methods and properties *are* part of `namedtuple`’s public interface. The helpers were named that way to avoid naming collisions with user-defined tuple fields. So go ahead and use them if you need them!

I want to show you a few scenarios where the `namedtuple` helper methods might come in handy. Let’s start with the `_asdict()` helper method. It returns the contents of a `namedtuple` as a dictionary:

```
>>> my_car._asdict()
OrderedDict([('color', 'red'), ('mileage', 3812.4)])
```

This is great for avoiding typos in the field names when generating JSON-output, for example:

```
>>> json.dumps(my_car._asdict())
'{"color": "red", "mileage": 3812.4}'
```

Another useful helper is the `_replace()` function. It creates a (shallow) copy of a tuple and allows you to selectively replace some of its fields:

```
>>> my_car._replace(color='blue')
Car(color='blue', mileage=3812.4)
```

Lastly, the `_make()` classmethod can be used to create new instances of a namedtuple from a sequence or iterable:

```
>>> Car._make(['red', 999])
Car(color='red', mileage=999)
```

## When to Use Namedtuples

Namedtuples can be an easy way to clean up your code and to make it more readable by enforcing a better structure for your data.

For example, I find that going from ad-hoc data types like dictionaries with a fixed format to namedtuples helps me express my intentions more clearly. Often when I attempt this refactoring I magically come up with a better solution for the problem I’m facing.

Using namedtuples over unstructured tuples and dicts can also make my coworkers’ lives easier because they make the data being passed around “self-documenting” (to a degree).

On the other hand, I try not to use namedtuples for their own sake if they don’t help me write “cleaner” and more maintainable code. Like many other techniques shown in this book, sometimes there can be *too much of a good thing*.

However, if you use them with care, namedtuples can undoubtedly make your Python code better and more expressive.

## Key Takeaways

- `collection.namedtuple` is a memory-efficient shortcut to manually define an immutable class in Python.
- Namedtuples can help clean up your code by enforcing an easier-to-understand structure on your data.

- Namedtuples provide a few useful helper methods that all start with a single underscore, but are part of the public interface. It's okay to use them.

## 4.7 Class vs Instance Variable Pitfalls

Besides making a distinction between class methods and instance methods, Python’s object model also distinguishes between class and instances *variables*.

It’s an important distinction, but also one that caused me trouble as a new Python developer. For a long time I didn’t invest the time needed to understand these concepts from the ground up. And so my early OOP experiments were riddled with surprising behaviors and odd bugs. In this chapter we’ll clear up any lingering confusion about this topic with some hands-on examples.

Like I said, there are two kinds of data attributes on Python objects: *class variables* and *instance variables*.

**Class variables** are declared inside the class definition (but outside of any instance methods). They’re not tied to any particular instance of a class. Instead, class variables store their contents on the class itself, and all objects created from a particular class share access to the same set of class variables. This means, for example, that modifying a class variable affects all object instances at the same time.

**Instance variables** are always tied to a particular object instance. Their contents are not stored on the class, but on each individual object created from the class. Therefore, the contents of an instance variable are completely independent from one object instance to the next. And so, modifying an instance variable only affects one object instance at a time.

Okay, this was fairly abstract—time to look at some code! Let’s bust out the old “dog example”... For some reason, OOP-tutorials always use cars or pets to illustrate their point, and it’s hard to break with that tradition.

What does a happy dog need? Four legs and a name:

```
class Dog:
    num_legs = 4 # <- Class variable

    def __init__(self, name):
        self.name = name # <- Instance variable
```

Alright, that's a neat object-oriented representation of the dog situation I just described. Creating new Dog instances works as expected, and they each get an instance variable called name:

```
>>> jack = Dog('Jack')
>>> jill = Dog('Jill')
>>> jack.name, jill.name
('Jack', 'Jill')
```

There's a little more flexibility when it comes to class variables. You can access the num\_legs class variable either directly on each Dog instance or *on the class itself*:

```
>>> jack.num_legs, jill.num_legs
(4, 4)
>>> Dog.num_legs
4
```

However, if you try to access an *instance* variable through the class, it'll fail with an `AttributeError`. Instance variables are specific to each object instance and are created when the `__init__` constructor runs—they don't even exist on the class itself.

This is the central distinction between class and instance variables:

```
>>> Dog.name
AttributeError:
"type object 'Dog' has no attribute 'name'"
```

Alright, so far so good.

Let's say that Jack the Dog gets a little too close to the microwave when he eats his dinner one day—and he sprouts an extra pair of legs. How'd you represent that in the little code sandbox we've got so far?

The first idea for a solution might be to simply modify the `num_legs` variable on the `Dog` class:

```
>>> Dog.num_legs = 6
```

But remember, we don't want *all* dogs to start scurrying around on six legs. So now we've just turned every dog instance in our little universe into Super Dog because we've modified a *class* variable. And this affects all dogs, even those created previously:

```
>>> jack.num_legs, jill.num_legs
(6, 6)
```

So that didn't work. The reason it didn't work is that modifying a class variable *on the class namespace* affects all instances of the class. Let's roll back the change to the class variable and instead try to give an extra pair o' legs specifically to Jack only:

```
>>> Dog.num_legs = 4
>>> jack.num_legs = 6
```

Now, what monstrosities did this create? Let's find out:

```
>>> jack.num_legs, jill.num_legs, Dog.num_legs
(6, 4, 4)
```

Okay, this looks “pretty good” (aside from the fact that we just gave poor Jack some extra legs). But how did this change actually affect our `Dog` objects?



You see, the trouble here is that while we got the result we wanted (extra legs for Jack), we introduced a `num_legs` instance variable to the Jack instance. And now the new `num_legs` instance variable “shadows” the class variable of the same name, overriding and hiding it when we access the object instance scope:

```
>>> jack.num_legs, jack.__class__.num_legs
(6, 4)
```

As you can see, the class variables seemingly got *out of sync*. This happened because writing to `jack.num_legs` created an *instance variable* with the same name as the class variable.

This isn’t necessarily bad, but it’s important to be aware of what happened here, behind the scenes. Before I finally understood class-level and instance-level scope in Python, this was a great avenue for bugs to slip into my programs.

To tell you the truth, trying to modify a class variable through an object instance—which then accidentally creates an instance variable of the same name, shadowing the original class variable—is a bit of an OOP pitfall in Python.

## A Dog-free Example

While no dogs were harmed in the making of this chapter (it’s all fun and games until someone sprouts an extra pair of legs), I wanted to give you one more practical example of the useful things you can do with class variables. Something that’s a little closer to the real-world applications for class variables.

So here it is. The following `CountedObject` class keeps track of how many times it was instantiated over the lifetime of a program (which might actually be an interesting performance metric to know):

```
class CountedObject:
    num_instances = 0

    def __init__(self):
        self.__class__.num_instances += 1
```

CountedObject keeps a `num_instances` class variable that serves as a shared counter. When the class is declared, it initializes the counter to zero and then leaves it alone.

Every time you create a new instance of this class, it increments the shared counter by one when the `__init__` constructor runs:

```
>>> CountedObject.num_instances
0
>>> CountedObject().num_instances
1
>>> CountedObject().num_instances
2
>>> CountedObject().num_instances
3
>>> CountedObject.num_instances
3
```

Notice how this code needs to jump through a little hoop to make sure it increments the counter variable *on the class*. It would've been an easy mistake to make if I had written the constructor as follows:

```
# WARNING: This implementation contains a bug

class BuggyCountedObject:
    num_instances = 0

    def __init__(self):
        self.num_instances += 1 # !!!
```

As you'll see, this (bad) implementation never increments the shared counter variable:

```
>>> BuggyCountedObject.num_instances
0
>>> BuggyCountedObject().num_instances
1
>>> BuggyCountedObject().num_instances
1
>>> BuggyCountedObject().num_instances
1
>>> BuggyCountedObject.num_instances
0
```

I'm sure you can see where I went wrong now. This (buggy) implementation never increments the shared counter because I made the mistake I explained in the “Jack the Dog” example earlier. This implementation won't work because I accidentally “shadowed” the `num_instance` class variable by creating an instance variable of the same name in the constructor.

It correctly calculates the new value for the counter (going from 0 to 1), but then stores the result in an instance variable—which means other instances of the class never even see the updated counter value.

As you can see, that's quite an easy mistake to make. It's a good idea to be careful and double-check your scoping when dealing with shared state on a class. Automated tests and peer reviews help greatly with that.

Nevertheless, I hope you can see why and how class variables—despite their pitfalls—can be useful tools in practice. Good luck!

## Key Takeaways

- Class variables are for data shared by all instances of a class. They belong to a class, not a specific instance and are shared

among all instances of a class.

- Instance variables are for data that is unique to each instance. They belong to individual object instances and are not shared among the other instances of a class. Each instance variable gets a unique backing store specific to the instance.
- Because class variables can be “shadowed” by instance variables of the same name, it’s easy to (accidentally) override class variables in a way that introduces bugs and odd behavior.

## 4.8 Instance, Class, and Static Methods Demystified

In this chapter you'll see what's behind *class methods*, *static methods*, and regular *instance methods* in Python.

If you develop an intuitive understanding for their differences, you'll be able to write object-oriented Python that communicates its intent more clearly and will be easier to maintain in the long run.

Let's begin by writing a (Python 3) class that contains simple examples for all three method types:

```
class MyClass:
    def method(self):
        return 'instance method called', self

    @classmethod
    def classmethod(cls):
        return 'class method called', cls

    @staticmethod
    def staticmethod():
        return 'static method called'
```

*Note for Python 2 users:* The `@staticmethod` and `@classmethod` decorators are available as of Python 2.4 and so this example will work as is. Instead of using a plain `class MyClass` declaration, you might choose to declare a new-style class inheriting from `object` with the `class MyClass(object)` syntax. But other than that, you're golden!

### Instance Methods

The first method on `MyClass`, called `method`, is a regular *instance method*. That's the basic, no-frills method type you'll use most of the time. You can see the method takes one parameter, `self`, which

points to an instance of `MyClass` when the method is called. But of course, instance methods can accept more than just one parameter.

Through the `self` parameter, instance methods can freely access attributes and other methods on the same object. This gives them a lot of power when it comes to modifying an object's state.

Not only can they modify object state, instance methods can also access the class itself through the `self.__class__` attribute. This means instance methods can also modify class state.

### Class Methods

Let's compare that to the second method, `MyClass.classmethod`. I marked this method with a `@classmethod`<sup>5</sup> decorator to flag it as a *class method*.

Instead of accepting a `self` parameter, class methods take a `cls` parameter that points to the class—and not the object instance—when the method is called.

Since the class method only has access to this `cls` argument, it can't modify object instance state. That would require access to `self`. However, class methods can still modify class state that applies across all instances of the class.

### Static Methods

The third method, `MyClass.staticmethod` was marked with a `@staticmethod`<sup>6</sup> decorator to flag it as a *static method*.

This type of method doesn't take a `self` or a `cls` parameter, although, of course, it can be made to accept an arbitrary number of other parameters.

---

<sup>5</sup>cf. Python Docs: "[@classmethod](#)"

<sup>6</sup>cf. Python Docs: "[@staticmethod](#)"

As a result, a static method cannot modify object state or class state. Static methods are restricted in what data they can access—they're primarily a way to namespace your methods.

### Let's See Them in Action!

I know this discussion has been fairly theoretical up to this point. I also believe it's important that you develop an intuitive understanding for how these method types differ in practice. That's why we'll go over some concrete examples now.

Let's take a look at how these methods behave in action when we call them. We'll start by creating an instance of the class and then calling the three different methods on it.

MyClass was set up in such a way that each method's implementation returns a tuple containing information we can use to trace what's going on and which parts of the class or object that method can access.

Here's what happens when we call an **instance method**:

```
>>> obj = MyClass()
>>> obj.method()
('instance method called', <MyClass instance at 0x11a2>)
```

This confirms that, in this case, the instance method called `method` has access to the object instance (printed as `<MyClass instance>`) via the `self` argument.

When the method is called, Python replaces the `self` argument with the instance object, `obj`. We could ignore the syntactic sugar provided by the `obj.method()` dot-call syntax and pass the instance object *manually* to get the same result:

```
>>> MyClass.method(obj)
('instance method called', <MyClass instance at 0x11a2>)
```

By the way, instance methods can also access the *class itself* through the `self.__class__` attribute. This makes instance methods powerful in terms of access restrictions—they can freely modify state on the object instance *and* on the class itself.

Let's try out the **class method** next:

```
>>> obj.classmethod()  
('class method called', <class MyClass at 0x11a2>)
```

Calling `classmethod()` showed us that it doesn't have access to the `<MyClass instance>` object, but only to the `<class MyClass>` object, representing the class itself (everything in Python is an object, even classes themselves).

Notice how Python automatically passes the class as the first argument to the function when we call `MyClass.classmethod()`. Calling a method in Python through the *dot syntax* triggers this behavior. The `self` parameter on instance methods works the same way.

Please note that naming these parameters `self` and `cls` is just a convention. You could just as easily name them `the_object` and `the_class` and get the same result. All that matters is that they're positioned first in the parameter list for that particular method.

Time to call the **static method** now:

```
>>> obj.staticmethod()  
'static method called'
```

Did you see how we called `staticmethod()` on the object and were able to do so successfully? Some developers are surprised when they learn that it's possible to call a static method on an object instance.

Behind the scenes, Python simply enforces the access restrictions by not passing in the `self` or the `cls` argument when a static method gets called using the dot syntax.



This confirms that static methods can neither access the object instance state nor the class state. They work like regular functions but belong to the class' (and every instance's) namespace.

Now, let's take a look at what happens when we attempt to call these methods on the class itself, without creating an object instance beforehand:

```
>>> MyClass.classmethod()
('class method called', <class MyClass at 0x11a2>)

>>> MyClass.staticmethod()
'static method called'

>>> MyClass.method()
TypeError: """unbound method method() must
    be called with MyClass instance as first
    argument (got nothing instead)"""
```

We were able to call `classmethod()` and `staticmethod()` just fine, but attempting to call the instance method `method()` failed with a `TypeError`.

This is to be expected. This time we didn't create an object instance and tried calling an instance function directly on the class blueprint itself. This means there is no way for Python to populate the `self` argument and therefore the call fails with a `TypeError` exception.

This should make the distinction between these three method types a little more clear. But don't worry, I'm not going to leave it at that. In the next two sections I'll go over two slightly more realistic examples of when to use these special method types.

I will base my examples around this bare-bones `Pizza` class:

```
class Pizza:
    def __init__(self, ingredients):
        self.ingredients = ingredients

    def __repr__(self):
        return f'Pizza({self.ingredients!r})'

>>> Pizza(['cheese', 'tomatoes'])
Pizza(['cheese', 'tomatoes'])
```

## Delicious Pizza Factories With @classmethod

If you've had any exposure to pizza in the real world, you'll know that there are many delicious variations available:

```
Pizza(['mozzarella', 'tomatoes'])
Pizza(['mozzarella', 'tomatoes', 'ham', 'mushrooms'])
Pizza(['mozzarella'] * 4)
```

The Italians figured out their pizza taxonomy centuries ago, and so these delicious types of pizza all have their own names. We'd do well to take advantage of that and give the users of our Pizza class a better interface for creating the pizza objects they crave.

A nice and clean way to do that is by using class methods as *factory functions*<sup>7</sup> for the different kinds of pizzas we can create:

```
class Pizza:
    def __init__(self, ingredients):
        self.ingredients = ingredients

    def __repr__(self):
        return f'Pizza({self.ingredients!r})'
```

---

<sup>7</sup>cf. Wikipedia: “Factory (object-oriented programming)”

```
@classmethod
def margherita(cls):
    return cls(['mozzarella', 'tomatoes'])

@classmethod
def prosciutto(cls):
    return cls(['mozzarella', 'tomatoes', 'ham'])
```

Note how I'm using the `cls` argument in the `margherita` and `prosciutto` factory methods instead of calling the `Pizza` constructor directly.

This is a trick you can use to follow the *Don't Repeat Yourself (DRY)*<sup>8</sup> principle. If we decide to rename this class at some point, we won't have to remember to update the constructor name in all of the factory functions.

Now, what can we do with these factory methods? Let's try them out:

```
>>> Pizza.margherita()
Pizza(['mozzarella', 'tomatoes'])

>>> Pizza.prosciutto()
Pizza(['mozzarella', 'tomatoes', 'ham'])
```

As you can see, we can use the factory functions to create new `Pizza` objects that are configured just the way we want them. They all use the same `__init__` constructor internally and simply provide a shortcut for remembering all of the various ingredients.

Another way to look at this use of class methods is to realize that they allow you to define alternative constructors for your classes.

Python only allows one `__init__` method per class. Using class methods makes it possible to add as many alternative constructors as nec-

---

<sup>8</sup>cf. Wikipedia: "Don't repeat yourself"

essary. This can make the interface for your classes self-documenting (to a certain degree) and simplify their usage.

## When To Use Static Methods

It's a little more difficult to come up with a good example here, but tell you what—I'll just keep stretching the pizza analogy thinner and thinner... (yum!)

Here's what I came up with:

```
import math

class Pizza:
    def __init__(self, radius, ingredients):
        self.radius = radius
        self.ingredients = ingredients

    def __repr__(self):
        return (f'Pizza({self.radius!r}, '
                f'{self.ingredients!r})')

    def area(self):
        return self.circle_area(self.radius)

    @staticmethod
    def circle_area(r):
        return r ** 2 * math.pi
```

Now what did I change here? First, I modified the constructor and `__repr__` to accept an extra radius argument.

I also added an `area()` instance method that calculates and returns the pizza's area. This would also be a good candidate for an `@property`—but hey, this is just a toy example.

Instead of calculating the area directly within `area()`, by using the well-known circle area formula, I factored that out to a separate `circle_area()` static method.

Let's try it out!

```
>>> p = Pizza(4, ['mozzarella', 'tomatoes'])
>>> p
Pizza(4, {self.ingredients})
>>> p.area()
50.26548245743669
>>> Pizza.circle_area(4)
50.26548245743669
```

Sure, this is still a bit of a simplistic example, but it'll help explain some of the benefits that static methods provide.

As we've learned, static methods can't access class or instance state because they don't take a `cls` or `self` argument. That's a big limitation—but it's also a great signal to show that a particular method is independent from everything else around it.

In the above example, it's clear that `circle_area()` can't modify the class or the class instance in any way. (Sure, you could always work around that with a global variable, but that's not the point here.)

Now, why is that useful?

Flagging a method as a static method is not just a hint that a method won't modify class or instance state. As you've seen, this restriction is also enforced by the Python runtime.

Techniques like that allow you to communicate clearly about parts of your class architecture so that new development work is naturally guided to happen within these boundaries. Of course, it would be easy enough to defy these restrictions. But in practice, they often help avoid accidental modifications that go against the original design.

Put differently, using static methods and class methods are ways to communicate developer intent while enforcing that intent enough to avoid most “slip of the mind” mistakes and bugs that would break the design.

Applied sparingly and when it makes sense, writing some of your methods that way can provide maintenance benefits and make it less likely that other developers use your classes incorrectly.

Static methods also have benefits when it comes to writing test code. Since the `circle_area()` method is completely independent from the rest of the class, it’s much easier to test.

We don’t have to worry about setting up a complete class instance before we can test the method in a unit test. We can just fire away like we would if we were testing a regular function. Again, this makes future maintenance easier and provides a link between object-oriented and procedural programming styles.

### Key Takeaways

- Instance methods need a class instance and can access the instance through `self`.
- Class methods don’t need a class instance. They can’t access the instance (`self`) but they have access to the class itself via `cls`.
- Static methods don’t have access to `cls` or `self`. They work like regular functions but belong to the class’ namespace.
- Static and class methods communicate and (to a certain degree) enforce developer intent about class design. This can have definite maintenance benefits.

# Chapter 5

## Common Data Structures in Python

What’s something that every Python developer should practice and learn more about?

Data structures. They’re the fundamental constructs around which you build your programs. Each data structure provides a particular way of organizing data so it can be accessed efficiently, depending on your use case.

I believe that going back to the fundamentals always pays off for a programmer, regardless of their skill level or experience.

Now, I don’t advocate that you should focus on expanding your data structures knowledge alone—the “failure mode” for that is getting stuck in theory la-la land and never shipping anything...

But I found that spending *some* time on brushing up your data structures (and algorithms) knowledge *always* pays off.

Whether you do that with a tightly focused “sprint” for a few days, or as an ongoing project with little pockets of time here and there doesn’t really matter. Either way, I promise it’ll be time well spent.

---

Alright, so data structures in Python, eh? We've got lists, dicts, sets...umm. Stacks? Do we have stacks?

You see, the trouble is that Python ships with an extensive set of data structures in its standard library. However, sometimes the naming for them is a bit “off”.

It's often unclear how even well-known “abstract data types” like a Stack correspond to a specific implementation in Python. Other languages like Java stick to a more “computer-sciency” and explicit naming scheme: A list isn't just a “list” in Java—it's either a `LinkedList` or an `ArrayList`.

This makes it easier to recognize the expected behavior and the computational complexity of these types. Python favors a simpler and more “human” naming scheme, and I love it. In part, it's what makes programming with Python so much fun.

But the downside is that even to experienced Python developers, it can be unclear whether the built-in `list` type is implemented as a linked list or a dynamic array. And the day will come when lacking this knowledge will cause them endless hours of frustration, or get them rejected in a job interview.

In this part of the book you'll take a tour of the fundamental data structures and implementations of abstract data types (ADTs) built into Python and its standard library.

My goal here is to clarify how the most common abstract data types map to Python's naming scheme and to provide a brief description for each. This information will also help you shine in Python coding interviews.

If you're looking for a good book to brush up on your general data structures knowledge, I highly recommend Steven S. Skiena's *The Algorithm Design Manual*.

It strikes a great balance between teaching you fundamental (and more advanced) data structures, and then showing you how to put



---

them to practical use in various algorithms. Steve's book was a great help in the writing of these chapters.

## 5.1 Dictionaries, Maps, and Hashtables

In Python, dictionaries (or “dicts” for short) are a central data structure. Dicts store an arbitrary number of objects, each identified by a unique dictionary *key*.

Dictionaries are also often called *maps*, *hashmaps*, *lookup tables*, or *associative arrays*. They allow for the efficient lookup, insertion, and deletion of any object associated with a given key.

What does this mean in practice? It turns out that *phone books* make a decent real-world analog for dictionary objects:

*Phone books allow you to quickly retrieve the information (phone number) associated with a given key (a person’s name). So, instead of having to read a phone book front to back in order to find someone’s number, you can jump more or less directly to a name and look up the associated information.*

This analogy breaks down somewhat when it comes to *how* the information is organized in order to allow for fast lookups. But the fundamental performance characteristics hold: Dictionaries allow you to quickly find the information associated with a given key.

In summary, dictionaries are one of the most frequently used and most important data structures in computer science.

So, how does Python handle dictionaries?

Let’s take a tour of the dictionary implementations available in core Python and the Python standard library.

### **dict – Your Go-To Dictionary**

Because of their importance, Python features a robust dictionary implementation that’s built directly into the core language: the `dict`

data type.<sup>1</sup>

Python also provides some useful “syntactic sugar” for working with dictionaries in your programs. For example, the curly-braces dictionary expression syntax and dictionary comprehensions allow you to conveniently define new dictionary objects:

```
phonebook = {
    'bob': 7387,
    'alice': 3719,
    'jack': 7052,
}

squares = {x: x * x for x in range(6)}

>>> phonebook['alice']
3719

>>> squares
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

There are some restrictions on which objects can be used as valid keys.

Python’s dictionaries are indexed by keys that can be of any hashable type<sup>2</sup>: A hashable object has a hash value which never changes during its lifetime (see `__hash__`), and it can be compared to other objects (see `__eq__`). In addition, hashable objects which compare as equal must have the same hash value.

Immutable types like strings and numbers are hashable and work well as dictionary keys. You can also use tuple objects as dictionary keys, as long as they contain only hashable types themselves.

For most use cases, Python’s built-in dictionary implementation will do everything you need. Dictionaries are highly optimized and un-

---

<sup>1</sup>cf. Python Docs: “Mapping Types — dict”

<sup>2</sup>cf. Python Docs Glossary: “Hashable”

derlie many parts of the language, for example class attributes and variables in a stack frame are both stored internally in dictionaries.

Python dictionaries are based on a well-tested and finely tuned hash table implementation that provides the performance characteristics you'd expect:  $O(1)$  time complexity for lookup, insert, update, and delete operations in the average case.

There's little reason not to use the standard `dict` implementation included with Python. However, specialized third-party dictionary implementations exist, for example skip lists or B-tree based dictionaries.

Besides “plain” `dict` objects, Python's standard library also includes a number of specialized dictionary implementations. These specialized dictionaries are all based on the built-in dictionary class (and share its performance characteristics), but add some convenience features on top of that.

Let's take a look at them.

## **`collections.OrderedDict` – Remember the Insertion Order of Keys**

Python includes a specialized `dict` subclass that remembers the insertion order of keys added to it: `collections.OrderedDict`.<sup>3</sup>

While standard `dict` instances preserve the insertion order of keys in CPython 3.6 and above, this is just a side effect of the CPython implementation and is not defined in the language spec.<sup>4</sup> So, if key order is important for your algorithm to work, it's best to communicate this clearly by explicitly using the `OrderDict` class.

By the way, `OrderedDict` is not a built-in part of the core language and must be imported from the `collections` module in the standard library.

---

<sup>3</sup>cf. Python Docs: “[collections.OrderedDict](#)”

<sup>4</sup>cf. CPython mailing list

```
>>> import collections
>>> d = collections.OrderedDict(one=1, two=2, three=3)

>>> d
OrderedDict([('one', 1), ('two', 2), ('three', 3)])

>>> d['four'] = 4
>>> d
OrderedDict([('one', 1), ('two', 2),
              ('three', 3), ('four', 4)])

>>> d.keys()
odict_keys(['one', 'two', 'three', 'four'])
```

## **collections.defaultdict – Return Default Values for Missing Keys**

The `defaultdict` class is another dictionary subclass that accepts a callable in its constructor whose return value will be used if a requested key cannot be found.<sup>5</sup>

This can save you some typing and make the programmer's intentions more clear, as compared to using the `get()` methods or catching a `KeyError` exception in regular dictionaries.

```
>>> from collections import defaultdict
>>> dd = defaultdict(list)

# Accessing a missing key creates it and
# initializes it using the default factory,
# i.e. list() in this example:
>>> dd['dogs'].append('Rufus')
>>> dd['dogs'].append('Kathrin')
>>> dd['dogs'].append('Mr Sniffles')
```

---

<sup>5</sup>cf. Python Docs: “[collections.defaultdict](#)”

```
>>> dd['dogs']  
['Rufus', 'Kathrin', 'Mr Sniffles']
```

## **collections.ChainMap – Search Multiple Dictionaries as a Single Mapping**

The `collections.ChainMap` data structure groups multiple dictionaries into a single mapping.<sup>6</sup> Lookups search the underlying mappings one by one until a key is found. Insertions, updates, and deletions only affect the first mapping added to the chain.

```
>>> from collections import ChainMap  
>>> dict1 = {'one': 1, 'two': 2}  
>>> dict2 = {'three': 3, 'four': 4}  
>>> chain = ChainMap(dict1, dict2)  
  
>>> chain  
ChainMap({'one': 1, 'two': 2}, {'three': 3, 'four': 4})  
  
# ChainMap searches each collection in the chain  
# from left to right until it finds the key (or fails):  
>>> chain['three']  
3  
>>> chain['one']  
1  
>>> chain['missing']  
KeyError: 'missing'
```

## **types.MappingProxyType – A Wrapper for Making Read-Only Dictionaries**

`MappingProxyType` is a wrapper around a standard dictionary that provides a read-only view into the wrapped dictionary's data.<sup>7</sup> This

---

<sup>6</sup>cf. Python Docs: “[collections.ChainMap](#)”

<sup>7</sup>cf. Python Docs: “[types.MappingProxyType](#)”

class was added in Python 3.3, and it can be used to create immutable proxy versions of dictionaries.

For example, this can be helpful if you'd like to return a dictionary carrying internal state from a class or module, while discouraging write access to this object. Using `MappingProxyType` allows you to put these restrictions in place without first having to create a full copy of the dictionary.

```
>>> from types import MappingProxyType
>>> writable = {'one': 1, 'two': 2}
>>> read_only = MappingProxyType(writable)

# The proxy is read-only:
>>> read_only['one']
1
>>> read_only['one'] = 23
TypeError:
"'mappingproxy' object does not support item assignment"

# Updates to the original are reflected in the proxy:
>>> writable['one'] = 42
>>> read_only
mappingproxy({'one': 42, 'two': 2})
```

## Dictionaries in Python: Conclusion

All of the Python dictionary implementations listed in this chapter are valid implementations that are built into the Python standard library.

If you're looking for a general recommendation on which mapping type to use in your programs, I'd point you to the built-in `dict` data type. It's a versatile and optimized hash table implementation that's built directly into the core language.

I would only recommend that you use one of the other data types listed here if you have special requirements that go beyond what's provided

by `dict`.

Yes, I still believe all of them are valid options—but usually your code will be more clear and easier to maintain by other developers if it relies on standard Python dictionaries most of the time.

### **Key Takeaways**

- Dictionaries are *the* central data structure in Python.
- The built-in `dict` type will be “good enough” most of the time.
- Specialized implementations, like read-only or ordered dicts, are available in the Python standard library.



## 5.2 Array Data Structures

An array is a fundamental data structure available in most programming languages, and it has a wide range of uses across different algorithms.

In this chapter we'll take a look at array implementations in Python that only use core language features or functionality that's included in the Python standard library.

You'll see the strengths and weaknesses of each approach so you can decide which implementation is right for your use case. But before we jump in, let's cover some of the basics first.

How do arrays work, and what are they used for?

Arrays consist of fixed-size data records that allow each element to be efficiently located based on its index.

Because arrays store information in adjoining blocks of memory, they're considered *contiguous* data structures (as opposed to *linked* data structure like linked lists, for example.)

A real world analogy for an array data structure is a *parking lot*:

*You can look at the parking lot as a whole and treat it as a single object, but inside the lot there are parking spots indexed by a unique number. Parking spots are containers for vehicles—each parking spot can either be empty or have a car, a motorbike, or some other vehicle parked on it.*

But not all parking lots are the same:

*Some parking lots may be restricted to only one type of vehicle. For example, a motor-home parking lot wouldn't allow bikes to be parked on it. A "restricted" parking lot*

*corresponds to a “typed array” data structure that only allows elements that have the same data type stored in them.*

Performance-wise, it’s very fast to look up an element contained in an array given the element’s index. A proper array implementation guarantees a constant  $O(1)$  access time for this case.

Python includes several array-like data structures in its standard library that each have slightly different characteristics. Let’s take a look at them:

## **list – Mutable Dynamic Arrays**

Lists are a part of the core Python language.<sup>8</sup> Despite their name, Python’s lists are implemented as *dynamic arrays* behind the scenes. This means a list allows elements to be added or removed, and the list will automatically adjust the backing store that holds these elements by allocating or releasing memory.

Python lists can hold arbitrary elements—“everything” is an object in Python, including functions. Therefore, you can mix and match different kinds of data types and store them all in a single list.

This can be a powerful feature, but the downside is that supporting multiple data types at the same time means that data is generally less tightly packed. And as a result, the whole structure takes up more space.

```
>>> arr = ['one', 'two', 'three']
>>> arr[0]
'one'

# Lists have a nice repr:
>>> arr
```

---

<sup>8</sup>cf. Python Docs: “list”

```
['one', 'two', 'three']

# Lists are mutable:
>>> arr[1] = 'hello'
>>> arr
['one', 'hello', 'three']

>>> del arr[1]
>>> arr
['one', 'three']

# Lists can hold arbitrary data types:
>>> arr.append(23)
>>> arr
['one', 'three', 23]
```

## tuple – Immutable Containers

Just like lists, tuples are also a part of the Python core language.<sup>9</sup> Unlike lists, however, Python’s tuple objects are immutable. This means elements can’t be added or removed dynamically—all elements in a tuple must be defined at creation time.

Just like lists, tuples can hold elements of arbitrary data types. Having this flexibility is powerful, but again, it also means that data is less tightly packed than it would be in a typed array.

```
>>> arr = 'one', 'two', 'three'
>>> arr[0]
'one'

# Tuples have a nice repr:
>>> arr
('one', 'two', 'three')
```

---

<sup>9</sup>cf. Python Docs: “tuple”

```

# Tuples are immutable:
>>> arr[1] = 'hello'
TypeError:
"'tuple' object does not support item assignment"

>>> del arr[1]
TypeError:
"'tuple' object doesn't support item deletion"

# Tuples can hold arbitrary data types:
# (Adding elements creates a copy of the tuple)
>>> arr + (23,)
('one', 'two', 'three', 23)

```

## array.array – Basic Typed Arrays

Python’s `array` module provides space-efficient storage of basic C-style data types like bytes, 32-bit integers, floating point numbers, and so on.

Arrays created with the `array.array` class are mutable and behave similarly to lists, except for one important difference—they are “typed arrays” constrained to a single data type.<sup>10</sup>

Because of this constraint, `array.array` objects with many elements are more space-efficient than lists and tuples. The elements stored in them are tightly packed, and this can be useful if you need to store many elements of the same type.

Also, arrays support many of the same methods as regular lists, and you might be able to use them as a “drop-in replacement” without requiring other changes to your application code.

```

>>> import array
>>> arr = array.array('f', (1.0, 1.5, 2.0, 2.5))

```

<sup>10</sup>cf. Python Docs: “`array.array`”

```
>>> arr[1]
1.5

# Arrays have a nice repr:
>>> arr
array('f', [1.0, 1.5, 2.0, 2.5])

# Arrays are mutable:
>>> arr[1] = 23.0
>>> arr
array('f', [1.0, 23.0, 2.0, 2.5])

>>> del arr[1]
>>> arr
array('f', [1.0, 2.0, 2.5])

>>> arr.append(42.0)
>>> arr
array('f', [1.0, 2.0, 2.5, 42.0])

# Arrays are "typed":
>>> arr[1] = 'hello'
TypeError: "must be real number, not str"
```

## str – Immutable Arrays of Unicode Characters

Python 3.x uses `str` objects to store textual data as immutable sequences of Unicode characters.<sup>11</sup> Practically speaking, that means a `str` is an immutable array of characters. Oddly enough, it's also a recursive data structure—each character in a string is a `str` object of length 1 itself.

String objects are space-efficient because they're tightly packed and they specialize in a single data type. If you're storing Unicode text, you should use them. Because strings are immutable in Python, modifying

---

<sup>11</sup>cf. Python Docs: "`str`"

a string requires creating a modified copy. The closest equivalent to a “mutable string” is storing individual characters inside a list.

```
>>> arr = 'abcd'
>>> arr[1]
'b'

>>> arr
'abcd'

# Strings are immutable:
>>> arr[1] = 'e'
TypeError:
"'str' object does not support item assignment"

>>> del arr[1]
TypeError:
"'str' object doesn't support item deletion"

# Strings can be unpacked into a list to
# get a mutable representation:
>>> list('abcd')
['a', 'b', 'c', 'd']
>>> ''.join(list('abcd'))
'abcd'

# Strings are recursive data structures:
>>> type('abc')
"<class 'str'>"
>>> type('abc'[0])
"<class 'str'>"
```

## bytes – Immutable Arrays of Single Bytes

Bytes objects are immutable sequences of single bytes (integers in the range of  $0 \leq x \leq 255$ ).<sup>12</sup> Conceptually, they’re similar to `str` objects, and you can also think of them as immutable arrays of bytes.

Like strings, bytes have their own literal syntax for creating objects and they’re space-efficient. Bytes objects are immutable, but unlike strings, there’s a dedicated “mutable byte array” data type called `bytearray` that they can be unpacked into. You’ll hear more about that in the next section.

```
>>> arr = bytes((0, 1, 2, 3))
>>> arr[1]
1

# Bytes literals have their own syntax:
>>> arr
b'x00x01x02x03'
>>> arr = b'x00x01x02x03'

# Only valid "bytes" are allowed:
>>> bytes((0, 300))
ValueError: "bytes must be in range(0, 256)"

# Bytes are immutable:
>>> arr[1] = 23
TypeError:
"'bytes' object does not support item assignment"

>>> del arr[1]
TypeError:
"'bytes' object doesn't support item deletion"
```

---

<sup>12</sup>cf. Python Docs: “bytes”

## bytearray – Mutable Arrays of Single Bytes

The `bytearray` type is a mutable sequence of integers in the range  $0 \leq x \leq 255$ .<sup>13</sup> They're closely related to `bytes` objects with the main difference being that `bytearrays` can be modified freely—you can overwrite elements, remove existing elements, or add new ones. The `bytearray` object will grow and shrink accordingly.

`Bytearrays` can be converted back into immutable `bytes` objects but this involves copying the stored data in full—a slow operation taking  $O(n)$  time.

```
>>> arr = bytearray((0, 1, 2, 3))
>>> arr[1]
1

# The bytearray repr:
>>> arr
bytearray(b'x00x01x02x03')

# Bytearrays are mutable:
>>> arr[1] = 23
>>> arr
bytearray(b'x00x17x02x03')

>>> arr[1]
23

# Bytearrays can grow and shrink in size:
>>> del arr[1]
>>> arr
bytearray(b'x00x02x03')

>>> arr.append(42)
>>> arr
bytearray(b'x00x02x03*')
```

<sup>13</sup>cf. Python Docs: “`bytearray`”



```
# Bytearrays can only hold "bytes"
# (integers in the range 0 <= x <= 255)
>>> arr[1] = 'hello'
TypeError: "an integer is required"

>>> arr[1] = 300
ValueError: "byte must be in range(0, 256)"

# Bytearrays can be converted back into bytes objects:
# (This will copy the data)
>>> bytes(arr)
b'x00x02x03*'
```

## Key Takeaways

There are a number of built-in data structures you can choose from when it comes to implementing arrays in Python. In this chapter we've focused on core language features and data structures included in the standard library only.

If you're willing to go beyond the Python standard library, third-party packages like *NumPy*<sup>14</sup> offer a wide range of fast array implementations for scientific computing and data science.

By restricting ourselves to the array data structures included with Python, here's what our choices come down to:

**You need to store arbitrary objects, potentially with mixed data types?** Use a list or a tuple, depending on whether you want an immutable data structure or not.

**You have numeric (integer or floating point) data and tight packing and performance is important?** Try out `array.array` and see if it does everything you need. Also, consider going beyond the standard library and try out packages like *NumPy* or *Pandas*.

---

<sup>14</sup>[www.numpy.org](http://www.numpy.org)

**You have textual data represented as Unicode characters?**

Use Python's built-in `str`. If you need a “mutable string,” use a list of characters.

**You want to store a contiguous block of bytes?** Use the immutable `bytes` type, or `bytearray` if you need a mutable data structure.

In most cases, I like to start out with a simple list. I'll only specialize later on if performance or storage space becomes an issue. Most of the time, using a general-purpose array data structure like list gives you the fastest development speed and the most programming convenience.

I found that this is usually much more important in the beginning than trying to squeeze out every last drop of performance right from the start.

## 5.3 Records, Structs, and Data Transfer Objects

Compared to arrays, record data structures provide a fixed number of fields, where each field can have a name and may also have a different type.

In this chapter, you'll see how to implement records, structs, and "plain old data objects" in Python, using only built-in data types and classes from the standard library.

By the way, I'm using the definition of a *record* loosely here. For example, I'm also going to discuss types like Python's built-in `tuple` that may or may not be considered records in a strict sense because they don't provide named fields.

Python offers several data types you can use to implement records, structs, and data transfer objects. In this chapter, you'll get a quick look at each implementation and its unique characteristics. At the end, you'll find a summary and a decision-making guide that will help you make your own picks.

Alright, let's get started!

### **dict – Simple Data Objects**

Python dictionaries store an arbitrary number of objects, each identified by a unique key.<sup>15</sup> Dictionaries are also often called *maps* or *associative arrays* and allow for the efficient lookup, insertion, and deletion of any object associated with a given key.

Using dictionaries as a record data type or data object in Python is possible. Dictionaries are easy to create in Python, as they have their own syntactic sugar built into the language in the form of dictionary literals. The dictionary syntax is concise and quite convenient to type.

---

<sup>15</sup>cf. "Dictionaries, Maps, and Hashtables" chapter

Data objects created using dictionaries are mutable, and there's little protection against misspelled field names, as fields can be added and removed freely at any time. Both of these properties can introduce surprising bugs, and there's always a trade-off to be made between convenience and error resilience.

```
car1 = {
    'color': 'red',
    'mileage': 3812.4,
    'automatic': True,
}
car2 = {
    'color': 'blue',
    'mileage': 40231,
    'automatic': False,
}

# Dicts have a nice repr:
>>> car2
{'color': 'blue', 'automatic': False, 'mileage': 40231}

# Get mileage:
>>> car2['mileage']
40231

# Dicts are mutable:
>>> car2['mileage'] = 12
>>> car2['windshield'] = 'broken'
>>> car2
{'windshield': 'broken', 'color': 'blue',
 'automatic': False, 'mileage': 12}

# No protection against wrong field names,
# or missing/extra fields:
car3 = {
    'colr': 'green',
```

```
'automatic': False,  
'windshield': 'broken',  
}
```

## tuple – Immutable Groups of Objects

Python’s tuples are simple data structures for grouping arbitrary objects.<sup>16</sup> Tuples are immutable—they cannot be modified once they’ve been created.

Performance-wise, tuples take up slightly less memory than lists in CPython,<sup>17</sup> and they’re also faster to construct.

As you can see in the bytecode disassembly below, constructing a tuple constant takes a single `LOAD_CONST` opcode, while constructing a list object with the same contents requires several more operations:

```
>>> import dis  
>>> dis.dis(compile("(23, 'a', 'b', 'c')", '', 'eval'))  
  0 LOAD_CONST                4 ((23, 'a', 'b', 'c'))  
  3 RETURN_VALUE  
  
>>> dis.dis(compile("[23, 'a', 'b', 'c']", '', 'eval'))  
  0 LOAD_CONST                0 (23)  
  3 LOAD_CONST                1 ('a')  
  6 LOAD_CONST                2 ('b')  
  9 LOAD_CONST                3 ('c')  
 12 BUILD_LIST                4  
 15 RETURN_VALUE
```

However, you shouldn’t place too much emphasis on these differences. In practice, the performance difference will often be negligible, and trying to squeeze extra performance out of a program by switching from lists to tuples will likely be the wrong approach.

---

<sup>16</sup>cf. Python Docs: “tuple”

<sup>17</sup>cf. CPython [tupleobject.c](#) and [listobject.c](#)

A potential downside of plain tuples is that the data you store in them can only be pulled out by accessing it through integer indexes. You can't give names to individual properties stored in a tuple. This can impact code readability.

Also, a tuple is always an ad-hoc structure: It's difficult to ensure that two tuples have the same number of fields and the same properties stored on them.

This makes it easy to introduce “slip-of-the-mind” bugs, such as mixing up the field order. Therefore, I would recommend that you keep the number of fields stored in a tuple as low as possible.

```
# Fields: color, mileage, automatic
>>> car1 = ('red', 3812.4, True)
>>> car2 = ('blue', 40231.0, False)

# Tuple instances have a nice repr:
>>> car1
('red', 3812.4, True)
>>> car2
('blue', 40231.0, False)

# Get mileage:
>>> car2[1]
40231.0

# Tuples are immutable:
>>> car2[1] = 12
TypeError:
"'tuple' object does not support item assignment"

# No protection against missing/extra fields
# or a wrong order:
>>> car3 = (3431.5, 'green', True, 'silver')
```

## Writing a Custom Class – More Work, More Control

Classes allow you to define reusable “blueprints” for data objects to ensure each object provides the same set of fields.

Using regular Python classes as record data types is feasible, but it also takes manual work to get the convenience features of other implementations. For example, adding new fields to the `__init__` constructor is verbose and takes time.

Also, the default string representation for objects instantiated from custom classes is not very helpful. To fix that you may have to add your own `__repr__` method,<sup>18</sup> which again is usually quite verbose and must be updated every time you add a new field.

Fields stored on classes are mutable, and new fields can be added freely, which you may or may not like. It’s possible to provide more access control and to create read-only fields using the `@property` decorator,<sup>19</sup> but once again, this requires writing more glue code.

Writing a custom class is a great option whenever you’d like to add business logic and *behavior* to your record objects using methods. However, this means that these objects are technically no longer plain data objects.

```
class Car:
    def __init__(self, color, mileage, automatic):
        self.color = color
        self.mileage = mileage
        self.automatic = automatic

>>> car1 = Car('red', 3812.4, True)
>>> car2 = Car('blue', 40231.0, False)
```

---

<sup>18</sup>cf. “String Conversion (Every Class Needs a `__repr__`)” chapter

<sup>19</sup>cf. [Python Docs: “property”](#)

```
# Get the mileage:
>>> car2.mileage
40231.0

# Classes are mutable:
>>> car2.mileage = 12
>>> car2.windshield = 'broken'

# String representation is not very useful
# (must add a manually written __repr__ method):
>>> car1
<Car object at 0x1081e69e8>
```

## **collections.namedtuple – Convenient Data Objects**

The `namedtuple` class available in Python 2.6+ provides an extension of the built-in `tuple` data type.<sup>20</sup> Similar to defining a custom class, using `namedtuple` allows you to define reusable “blueprints” for your records that ensure the correct field names are used.

Namedtuples are immutable, just like regular tuples. This means you cannot add new fields or modify existing fields after the `namedtuple` instance was created.

Besides that, `namedtuples` are, well... named tuples. Each object stored in them can be accessed through a unique identifier. This frees you from having to remember integer indexes, or resort to workarounds like defining integer constants as mnemonics for your indexes.

`Namedtuple` objects are implemented as regular Python classes internally. When it comes to memory usage, they are also “better” than regular classes and just as memory efficient as regular tuples:

---

<sup>20</sup>cf. “What Namedtuples Are Good For” chapter



```
>>> from collections import namedtuple
>>> from sys import getsizeof

>>> p1 = namedtuple('Point', 'x y z')(1, 2, 3)
>>> p2 = (1, 2, 3)

>>> getsizeof(p1)
72
>>> getsizeof(p2)
72
```

Namedtuples can be an easy way to clean up your code and make it more readable by enforcing a better structure for your data.

I find that going from ad-hoc data types, like dictionaries with a fixed format, to namedtuples helps me express the intent of my code more clearly. Often when I apply this refactoring, I magically come up with a better solution for the problem I'm facing.

Using namedtuples over regular (unstructured) tuples and dicts can also make my coworkers' lives easier: Namedtuples make the data that's being passed around “self-documenting”, at least to a degree.

```
>>> from collections import namedtuple
>>> Car = namedtuple('Car' , 'color mileage automatic')
>>> car1 = Car('red', 3812.4, True)

# Instances have a nice repr:
>>> car1
Car(color='red', mileage=3812.4, automatic=True)

# Accessing fields:
>>> car1.mileage
3812.4

# Fields are immutable:
```

```
>>> car1.mileage = 12
AttributeError: "can't set attribute"
>>> car1.windshield = 'broken'
AttributeError:
"'Car' object has no attribute 'windshield'"
```

## **typing.NamedTuple – Improved Namedtuples**

This class added in Python 3.6 is the younger sibling of the `namedtuple` class in the `collections` module.<sup>21</sup> It is very similar to `namedtuple`, the main difference being an updated syntax for defining new record types and added support for type hints.

Please note that type annotations are not enforced without a separate type-checking tool like *mypy*.<sup>22</sup> But even without tool support, they can provide useful hints for other programmers (or be terribly confusing if the type hints become out-of-date.)

```
>>> from typing import NamedTuple

class Car(NamedTuple):
    color: str
    mileage: float
    automatic: bool

>>> car1 = Car('red', 3812.4, True)

# Instances have a nice repr:
>>> car1
Car(color='red', mileage=3812.4, automatic=True)

# Accessing fields:
>>> car1.mileage
3812.4
```

---

<sup>21</sup>cf. Python Docs: “[typing.NamedTuple](#)”

<sup>22</sup>[mypy-lang.org](http://mypy-lang.org)

```
# Fields are immutable:
>>> car1.mileage = 12
AttributeError: "can't set attribute"
>>> car1.windshield = 'broken'
AttributeError:
"'Car' object has no attribute 'windshield'"

# Type annotations are not enforced without
# a separate type checking tool like mypy:
>>> Car('red', 'NOT_A_FLOAT', 99)
Car(color='red', mileage='NOT_A_FLOAT', automatic=99)
```

## **struct.Struct – Serialized C Structs**

The `struct.Struct` class<sup>23</sup> converts between Python values and C structs serialized into Python bytes objects. For example, it can be used to handle binary data stored in files or coming in from network connections.

Structs are defined using a format strings-like mini language that allows you to define the arrangement of various C data types like `char`, `int`, and `long`, as well as their unsigned variants.

Serialized structs are seldom used to represent data objects meant to be handled purely inside Python code. They're intended primarily as a data exchange format, rather than as a way of holding data in memory that's only used by Python code.

In some cases, packing primitive data into structs may use less memory than keeping it in other data types. However, in most cases that would be quite an advanced (and probably unnecessary) optimization.

```
>>> from struct import Struct
>>> MyStruct = Struct('i?f')
>>> data = MyStruct.pack(23, False, 42.0)
```

---

<sup>23</sup>cf. Python Docs: “`struct.Struct`”

```
# All you get is a blob of data:
>>> data
b'x17x00x00x00x00x00x00x00x00(B'

# Data blobs can be unpacked again:
>>> MyStruct.unpack(data)
(23, False, 42.0)
```

## **types.SimpleNamespace – Fancy Attribute Access**

Here’s one more “esoteric” choice for implementing data objects in Python: `types.SimpleNamespace`.<sup>24</sup> This class was added in Python 3.3 and it provides attribute access to its namespace.

This means `SimpleNamespace` instances expose all of their keys as class attributes. This means you can use `obj.key` “dotted” attribute access instead of the `obj['key']` square-brackets indexing syntax that’s used by regular dicts. All instances also include a meaningful `__repr__` by default.

As its name proclaims, `SimpleNamespace` is simple! It’s basically a glorified dictionary that allows attribute access and prints nicely. Attributes can be added, modified, and deleted freely.

```
>>> from types import SimpleNamespace
>>> car1 = SimpleNamespace(color='red',
...                         mileage=3812.4,
...                         automatic=True)

# The default repr:
>>> car1
namespace(automatic=True, color='red', mileage=3812.4)

# Instances support attribute access and are mutable:
```

---

<sup>24</sup>cf. Python Docs: “`types.SimpleNamespace`”

```
>>> car1.mileage = 12
>>> car1.windshield = 'broken'
>>> del car1.automatic
>>> car1
namespace(color='red', mileage=12, windshield='broken')
```

## Key Takeaways

Now, which type should you use for data objects in Python? As you've seen, there's quite a number of different options for implementing records or data objects. Generally your decision will depend on your use case:

**You only have a few (2-3) fields:** Using a plain tuple object may be okay if the field order is easy to remember or field names are superfluous. For example, think of an (x, y, z) point in 3D space.

**You need immutable fields:** In this case, plain tuples, `collections.namedtuple`, and `typing.NamedTuple` would all make good options for implementing this type of data object.

**You need to lock down field names to avoid typos:** `collections.namedtuple` and `typing.NamedTuple` are your friends here.

**You want to keep things simple:** A plain dictionary object might be a good choice due to the convenient syntax that closely resembles JSON.

**You need full control over your data structure:** It's time to write a custom class with `@property` setters and getters.

**You need to add behavior (methods) to the object:** You should write a custom class, either from scratch or by extending `collections.namedtuple` or `typing.NamedTuple`.

**You need to pack data tightly to serialize it to disk or to send it over the network:** Time to read up on `struct.Struct` because

this is a great use case for it.

If you're looking for a safe default choice, my general recommendation for implementing a plain record, struct, or data object in Python would be to use `collections.namedtuple` in Python 2.x and its younger sibling, `typing.NamedTuple` in Python 3.

## 5.4 Sets and Multisets

In this chapter you'll see how to implement mutable and immutable set and multiset (bag) data structures in Python, using built-in data types and classes from the standard library. First though, let's do a quick recap of what a set data structure is:

A *set* is an unordered collection of objects that does not allow duplicate elements. Typically, sets are used to quickly test a value for membership in the set, to insert or delete new values from a set, and to compute the union or intersection of two sets.

In a “proper” set implementation, membership tests are expected to run in fast  $O(1)$  time. Union, intersection, difference, and subset operations should take  $O(n)$  time on average. The set implementations included in Python's standard library follow these performance characteristics.<sup>25</sup>

Just like dictionaries, sets get special treatment in Python and have some syntactic sugar that makes them easy to create. For example, the curly-braces set expression syntax and set comprehensions allow you to conveniently define new set instances:

```
vowels = {'a', 'e', 'i', 'o', 'u'}  
squares = {x * x for x in range(10)}
```

But be careful: To create *an empty set* you'll need to call the `set()` constructor. Using empty curly-braces `{}` is ambiguous and will create an empty dictionary instead.

Python and its standard library provide several set implementations. Let's have a look at them.

---

<sup>25</sup>cf. [wiki.python.org/moin/TimeComplexity](http://wiki.python.org/moin/TimeComplexity)

## set – Your Go-To Set

This is the built-in set implementation in Python.<sup>26</sup> The `set` type is mutable and allows for the dynamic insertion and deletion of elements.

Python’s sets are backed by the `dict` data type and share the same performance characteristics. Any hashable object can be stored in a set.<sup>27</sup>

```
>>> vowels = {'a', 'e', 'i', 'o', 'u'}
>>> 'e' in vowels
True

>>> letters = set('alice')
>>> letters.intersection(vowels)
{'a', 'e', 'i'}

>>> vowels.add('x')
>>> vowels
{'i', 'a', 'u', 'o', 'x', 'e'}

>>> len(vowels)
6
```

## frozenset – Immutable Sets

The `frozenset` class implements an *immutable* version of `set` that cannot be changed after it has been constructed.<sup>28</sup> `Frozensets` are static and only allow query operations on their elements (no inserts or deletions.) Because `frozensets` are static and hashable, they can be used as dictionary keys or as elements of another set, something that isn’t possible with regular (mutable) set objects.

---

<sup>26</sup>cf. Python Docs: “set”

<sup>27</sup>cf. Python Docs: “hashable”

<sup>28</sup>cf. Python Docs: “frozenset”



```
>>> vowels = frozenset({'a', 'e', 'i', 'o', 'u'})
>>> vowels.add('p')
AttributeError:
"'frozenset' object has no attribute 'add'"

# Frozensets are hashable and can
# be used as dictionary keys:
>>> d = { frozenset({1, 2, 3}): 'hello' }
>>> d[frozenset({1, 2, 3})]
'hello'
```

## **collections.Counter – Multisets**

The `collections.Counter` class in the Python standard library implements a multiset (or bag) type that allows elements in the set to have more than one occurrence.<sup>29</sup>

This is useful if you need to keep track of not only *if* an element is part of a set, but also *how many times* it is included in the set:

```
>>> from collections import Counter
>>> inventory = Counter()

>>> loot = {'sword': 1, 'bread': 3}
>>> inventory.update(loot)
>>> inventory
Counter({'bread': 3, 'sword': 1})

>>> more_loot = {'sword': 1, 'apple': 1}
>>> inventory.update(more_loot)
>>> inventory
Counter({'bread': 3, 'sword': 2, 'apple': 1})
```

Here's a caveat for the `Counter` class: You'll want to be careful when counting the number of elements in a `Counter` object. Calling `len()`

<sup>29</sup>cf. Python Docs: "[collections.Counter](#)"

returns the number of *unique* elements in the multiset, whereas the total number of elements can be retrieved using the `sum` function:

```
>>> len(inventory)
3  # Unique elements

>>> sum(inventory.values())
6  # Total no. of elements
```

## Key Takeaways

- Sets are another useful and commonly used data structure included with Python and its standard library.
- Use the built-in `set` type when looking for a mutable set.
- `frozenset` objects are hashable and can be used as dictionary or set keys.
- `collections.Counter` implements multiset or “bag” data structures.

## 5.5 Stacks (LIFOs)

A stack is a collection of objects that supports fast *last-in, first-out* (*LIFO*) semantics for inserts and deletes. Unlike lists or arrays, stacks typically don't allow for random access to the objects they contain. The insert and delete operations are also often called *push* and *pop*.

A useful real-world analogy for a stack data structure is a *stack of plates*:

*New plates are added to the top of the stack. And because the plates are precious and heavy, only the topmost plate can be moved (last-in, first-out). To reach the plates that are lower down in the stack, the topmost plates must be removed one by one.*

Stacks and queues are similar. They're both linear collections of items, and the difference lies in the order that the items are accessed:

With a **queue**, you remove the item *least* recently added (*first-in, first-out* or *FIFO*); but with a **stack**, you remove the item *most* recently added (*last-in, first-out* or *LIFO*).

Performance-wise, a proper stack implementation is expected to take  $O(1)$  time for insert and delete operations.

Stacks have a wide range of uses in algorithms, for example, in language parsing and runtime memory management ("call stack"). A short and beautiful algorithm using a stack is depth-first search (DFS) on a tree or graph data structure.

Python ships with several stack implementations that each have slightly different characteristics. We'll now take a look at them and compare their characteristics.

## list – Simple, Built-In Stacks

Python’s built-in `list` type makes a decent stack data structure as it supports push and pop operations in amortized  $O(1)$  time.<sup>30</sup>

Python’s lists are implemented as dynamic arrays internally, which means they occasionally need to resize the storage space for elements stored in them when elements are added or removed. The list over-allocates its backing storage so that not every push or pop requires resizing, and as a result, you get an amortized  $O(1)$  time complexity for these operations.

The downside is that this makes their performance less consistent than the stable  $O(1)$  inserts and deletes provided by a linked list based implementation (like `collections.deque`, see below). On the other hand, lists do provide fast  $O(1)$  time random access to elements on the stack, and this can be an added benefit.

Here’s an important performance caveat you should be aware of when using lists as stacks:

To get the amortized  $O(1)$  performance for inserts and deletes, new items must be added to the *end* of the list with the `append()` method and removed again from the end using `pop()`. For optimum performance, stacks based on Python lists should grow towards higher indexes and shrink towards lower ones.

Adding and removing from the front is much slower and takes  $O(n)$  time, as the existing elements must be shifted around to make room for the new element. This is a performance antipattern that you should avoid as much as possible.

```
>>> s = []
>>> s.append('eat')
>>> s.append('sleep')
>>> s.append('code')
```

---

<sup>30</sup>cf. Python Docs: “Using lists as stacks”

```
>>> s
['eat', 'sleep', 'code']

>>> s.pop()
'code'
>>> s.pop()
'sleep'
>>> s.pop()
'eat'

>>> s.pop()
IndexError: "pop from empty list"
```

## **collections.deque – Fast & Robust Stacks**

The deque class implements a double-ended queue that supports adding and removing elements from either end in  $O(1)$  time (non-amortized). Because deques support adding and removing elements from either end equally well, they can serve both as queues and as stacks.<sup>31</sup>

Python's deque objects are implemented as doubly-linked lists which gives them excellent and consistent performance for inserting and deleting elements, but poor  $O(n)$  performance for randomly accessing elements in the middle of a stack.<sup>32</sup>

Overall, `collections.deque` is a great choice if you're looking for a stack data structure in Python's standard library that has the performance characteristics of a linked-list implementation.

```
>>> from collections import deque
>>> s = deque()
>>> s.append('eat')
>>> s.append('sleep')
```

---

<sup>31</sup>cf. Python Docs: “[collections.deque](#)”

<sup>32</sup>cf. CPython `_collectionsmodule.c`

```
>>> s.append('code')

>>> s
deque(['eat', 'sleep', 'code'])

>>> s.pop()
'code'
>>> s.pop()
'sleep'
>>> s.pop()
'eat'

>>> s.pop()
IndexError: "pop from an empty deque"
```

## queue.LifoQueue – Locking Semantics for Parallel Computing

This stack implementation in the Python standard library is synchronized and provides locking semantics to support multiple concurrent producers and consumers.<sup>33</sup>

Besides LifoQueue, the queue module contains several other classes that implement multi-producer/multi-consumer queues that are useful for parallel computing.

Depending on your use case, the locking semantics might be helpful, or they might just incur unneeded overhead. In this case you'd be better off with using a list or a deque as a general-purpose stack.

```
>>> from queue import LifoQueue
>>> s = LifoQueue()
>>> s.put('eat')
>>> s.put('sleep')
>>> s.put('code')
```

---

<sup>33</sup>cf. Python Docs: “queue.LifoQueue”

```
>>> s
<queue.LifoQueue object at 0x108298dd8>

>>> s.get()
'code'
>>> s.get()
'sleep'
>>> s.get()
'eat'

>>> s.get_nowait()
queue.Empty

>>> s.get()
# Blocks / waits forever...
```

## Comparing Stack Implementations in Python

As you’ve seen, Python ships with several implementations for a stack data structure. All of them have slightly different characteristics, as well as performance and usage trade-offs.

If you’re not looking for parallel processing support (or don’t want to handle locking and unlocking manually), your choice comes down to the built-in `list` type or `collections.deque`. The difference lies in the data structure used behind the scenes and overall ease of use:

- `list` is backed by a dynamic array which makes it great for fast random access, but requires occasional resizing when elements are added or removed. The list over-allocates its backing storage so that not every push or pop requires resizing, and you get an amortized  $O(1)$  time complexity for these operations. But you do need to be careful to only insert and remove items “from the right side” using `append()` and `pop()`. Otherwise, performance slows down to  $O(n)$ .

- `collections.deque` is backed by a doubly-linked list which optimizes appends and deletes at both ends and provides consistent  $O(1)$  performance for these operations. Not only is its performance more stable, the deque class is also easier to use because you don't have to worry about adding or removing items from "the wrong end."

In summary, I believe that `collections.deque` is an excellent choice for implementing a stack (LIFO queue) in Python.

## Key Takeaways

- Python ships with several stack implementations that have slightly different performance and usage characteristics.
- `collections.deque` provides a safe and fast general-purpose stack implementation.
- The built-in `list` type can be used as a stack, but be careful to only append and remove items with `append()` and `pop()` in order to avoid slow performance.



## 5.6 Queues (FIFOs)

In this chapter you'll see how to implement a FIFO queue data structure using only built-in data types and classes from the Python standard library. But first, let's recap what a queue is:

A queue is a collection of objects that supports fast *first-in, first-out (FIFO)* semantics for inserts and deletes. The insert and delete operations are sometimes called *enqueue* and *dequeue*. Unlike lists or arrays, queues typically don't allow for random access to the objects they contain.

Here's a real-world analogy for a first-in, first-out queue:

*Imagine a line of Pythonistas waiting to pick up their conference badges on day one of PyCon registration. New additions to the line are made to the back of the queue as new people enter the conference venue and “queue up” to receive their badges. Removal (serving) happens in the front of the queue, as developers receive their badges and conference swag bags and leave the queue.*

Another way to memorize the characteristics of a queue data structure is to think of it as a *pipe*:

*New items (water molecules, ping-pong balls, ...) are put in at one end and travel to the other where you or someone else removes them again. While the items are in the queue (a solid metal pipe) you can't get at them. The only way to interact with the items in the queue is to add new items at the back (enqueue) or to remove items at the front (dequeue) of the pipe.*

Queues are similar to stacks, and the difference between them lies in how items are removed:

With a **queue**, you remove the item *least* recently added (*first-in, first-out* or *FIFO*); but with a **stack**, you remove the item *most* recently added (*last-in, first-out* or *LIFO*).

Performance-wise, a proper queue implementation is expected to take  $O(1)$  time for insert and delete operations. These are the two main operations performed on a queue, and in a correct implementation, they should be fast.

Queues have a wide range of applications in algorithms and often help solve scheduling and parallel programming problems. A short and beautiful algorithm using a queue is breadth-first search (BFS) on a tree or graph data structure.

Scheduling algorithms often use priority queues internally. These are specialized queues: Instead of retrieving the next element by insertion time, a priority queue retrieves the *highest-priority* element. The priority of individual elements is decided by the queue, based on the ordering applied to their keys. We'll take a closer look at priority queues and how they're implemented in Python in the next chapter.

A regular queue, however, won't re-order the items it carries. Just like in the pipe example, "you'll get what you put in" and in exactly that order.

Python ships with several queue implementations that each have slightly different characteristics. Let's review them.

## list — Terribly Sloooow Queues

It's possible to use a regular `list` as a queue but this is not ideal from a performance perspective.<sup>34</sup> Lists are quite slow for this purpose because inserting or deleting an element at the beginning requires shifting all of the other elements by one, requiring  $O(n)$  time.

Therefore, I would *not recommend* using a `list` as a makeshift queue in Python (unless you're only dealing with a small number of

---

<sup>34</sup>cf. Python Docs: "Using lists as queues"

elements).

```
>>> q = []
>>> q.append('eat')
>>> q.append('sleep')
>>> q.append('code')

>>> q
['eat', 'sleep', 'code']

# Careful: This is slow!
>>> q.pop(0)
'eat'
```

## **collections.deque – Fast & Robust Queues**

The deque class implements a double-ended queue that supports adding and removing elements from either end in  $O(1)$  time (non-amortized). Because deques support adding and removing elements from either end equally well, they can serve both as queues and as stacks.<sup>35</sup>

Python’s deque objects are implemented as doubly-linked lists.<sup>36</sup> This gives them excellent and consistent performance for inserting and deleting elements, but poor  $O(n)$  performance for randomly accessing elements in the middle of the stack.

As a result, `collections.deque` is a great default choice if you’re looking for a queue data structure in Python’s standard library.

```
>>> from collections import deque
>>> q = deque()
>>> q.append('eat')
>>> q.append('sleep')
```

---

<sup>35</sup>cf. [Python Docs: “collections.deque”](#)

<sup>36</sup>cf. `CPython_collectionsmodule.c`

```
>>> q.append('code')

>>> q
deque(['eat', 'sleep', 'code'])

>>> q.popleft()
'eat'
>>> q.popleft()
'sleep'
>>> q.popleft()
'code'

>>> q.popleft()
IndexError: "pop from an empty deque"
```

## queue.Queue – Locking Semantics for Parallel Computing

This queue implementation in the Python standard library is synchronized and provides locking semantics to support multiple concurrent producers and consumers.<sup>37</sup>

The queue module contains several other classes implementing multi-producer/multi-consumer queues that are useful for parallel computing.

Depending on your use case, the locking semantics might be helpful or just incur unneeded overhead. In this case, you'd be better off using `collections.deque` as a general-purpose queue.

```
>>> from queue import Queue
>>> q = Queue()
>>> q.put('eat')
>>> q.put('sleep')
>>> q.put('code')
```

---

<sup>37</sup>cf. Python Docs: “[queue.Queue](#)”

```
>>> q
<queue.Queue object at 0x1070f5b38>

>>> q.get()
'eat'
>>> q.get()
'sleep'
>>> q.get()
'code'

>>> q.get_nowait()
queue.Empty

>>> q.get()
# Blocks / waits forever...
```

## **multiprocessing.Queue – Shared Job Queues**

This is a shared job queue implementation that allows queued items to be processed in parallel by multiple concurrent workers.<sup>38</sup> Process-based parallelization is popular in CPython due to the global interpreter lock (GIL) that prevents some forms of parallel execution on a single interpreter process.

As a specialized queue implementation meant for sharing data between processes, `multiprocessing.Queue` makes it easy to distribute work across multiple processes in order to work around the GIL limitations. This type of queue can store and transfer any pickle-able object across process boundaries.

```
>>> from multiprocessing import Queue
>>> q = Queue()
>>> q.put('eat')
>>> q.put('sleep')
```

---

<sup>38</sup>cf. Python Docs: “[multiprocessing.Queue](#)”

```
>>> q.put('code')

>>> q
<multiprocessing.queues.Queue object at 0x1081c12b0>

>>> q.get()
'eat'
>>> q.get()
'sleep'
>>> q.get()
'code'

>>> q.get()
# Blocks / waits forever...
```

## Key Takeaways

- Python includes several queue implementations as part of the core language and its standard library.
- list objects can be used as queues, but this is generally not recommended due to slow performance.
- If you're not looking for parallel processing support, the implementation offered by `collections.deque` is an excellent default choice for implementing a FIFO queue data structure in Python. It provides the performance characteristics you'd expect from a good queue implementation and can also be used as a stack (LIFO Queue).

## 5.7 Priority Queues

A priority queue is a container data structure that manages a set of records with totally-ordered<sup>39</sup> keys (for example, a numeric *weight* value) to provide quick access to the record with the *smallest* or *largest* key in the set.

You can think of a priority queue as a modified queue: instead of retrieving the next element by insertion time, it retrieves the *highest-priority* element. The priority of individual elements is decided by the ordering applied to their keys.

Priority queues are commonly used for dealing with scheduling problems, for example, to give precedence to tasks with higher urgency.

Think about the job of an operating system task scheduler:

*Ideally, high-priority tasks on the system (e.g., playing a real-time game) should take precedence over lower-priority tasks (e.g., downloading updates in the background). By organizing pending tasks in a priority queue that uses the task urgency as the key, the task scheduler can quickly select the highest-priority tasks and allow them to run first.*

In this chapter you'll see a few options for how you can implement Priority Queues in Python using built-in data structures or data structures that ship with Python's standard library. Each implementation will have their own upsides and downsides, but in my mind there's a clear winner for most common scenarios. Let's find out which one it is.

### **list – Maintaining a Manually Sorted Queue**

You can use a sorted list to quickly identify and delete the smallest or largest element. The downside is that inserting new elements into

---

<sup>39</sup>cf. Wikipedia “Total order”

a list is a slow  $O(n)$  operation.

While the insertion point can be found in  $O(\log n)$  time using `bisect.insort`<sup>40</sup> in the standard library, this is always dominated by the slow insertion step.

Maintaining the order by appending to the list and re-sorting also takes at least  $O(n \log n)$  time. Another downside is that you must manually take care of re-sorting the list when new elements are inserted. It's easy to introduce bugs by missing this step, and the burden is always on you, the developer.

Therefore, I believe that sorted lists are only suitable as priority queues when there will be few insertions.

```
q = []

q.append((2, 'code'))
q.append((1, 'eat'))
q.append((3, 'sleep'))

# NOTE: Remember to re-sort every time
#       a new element is inserted, or use
#       bisect.insort().
q.sort(reverse=True)

while q:
    next_item = q.pop()
    print(next_item)

# Result:
# (1, 'eat')
# (2, 'code')
# (3, 'sleep')
```

---

<sup>40</sup>cf. Python Docs: “bisect.insort”



## heapq – List-Based Binary Heaps

This is a binary heap implementation usually backed by a plain list, and it supports insertion and extraction of the smallest element in  $O(\log n)$  time.<sup>41</sup>

This module is a good choice for implementing priority queues in Python. Since `heapq` technically only provides a min-heap implementation, extra steps must be taken to ensure sort stability and other features typically expected from a “practical” priority queue.<sup>42</sup>

```
import heapq

q = []

heapq.heappush(q, (2, 'code'))
heapq.heappush(q, (1, 'eat'))
heapq.heappush(q, (3, 'sleep'))

while q:
    next_item = heapq.heappop(q)
    print(next_item)

# Result:
# (1, 'eat')
# (2, 'code')
# (3, 'sleep')
```

## queue.PriorityQueue – Beautiful Priority Queues

This priority queue implementation uses `heapq` internally and shares the same time and space complexities.<sup>43</sup>

---

<sup>41</sup>cf. Python Docs: “`heapq`”

<sup>42</sup>cf. Python Docs: “`heapq` – Priority queue implementation notes”

<sup>43</sup>cf. Python Docs: “`queue.PriorityQueue`”

The difference is that `PriorityQueue` is synchronized and provides locking semantics to support multiple concurrent producers and consumers.

Depending on your use case, this might be helpful—or just slow your program down slightly. In any case, you might prefer the class-based interface provided by `PriorityQueue` over using the function-based interface provided by `heapq`.

```
from queue import PriorityQueue

q = PriorityQueue()

q.put((2, 'code'))
q.put((1, 'eat'))
q.put((3, 'sleep'))

while not q.empty():
    next_item = q.get()
    print(next_item)

# Result:
# (1, 'eat')
# (2, 'code')
# (3, 'sleep')
```

## Key Takeaways

- Python includes several priority queue implementations for you to use.
- `queue.PriorityQueue` stands out from the pack with a nice object-oriented interface and a name that clearly states its intent. It should be your preferred choice.
- If you'd like to avoid the locking overhead of `queue.PriorityQueue`, using the `heapq` module directly is also a good option.

# **Chapter 6**

## **Looping & Iteration**

## 6.1 Writing Pythonic Loops

One of the easiest ways to spot a developer with a background in C-style languages who only recently picked up Python is to look at how they write loops.

For example, whenever I see a code snippet like the following, that's an example of someone trying to write Python like it's C or Java:

```
my_items = ['a', 'b', 'c']

i = 0
while i < len(my_items):
    print(my_items[i])
    i += 1
```

Now, what's so “unpythonic” about this code, you ask? Two things:

First, it keeps track of the index `i` manually—initializing, it to zero and then carefully incrementing it upon every loop iteration.

And second, it uses `len()` to get the size of the `my_items` container in order to determine how often to iterate.

In Python you can write loops that handle both of these responsibilities automatically. It's a great idea to take advantage of that. For example, it's much harder to write accidental infinite loops if your code doesn't have to keep track of a running index. It also makes the code more concise and therefore more readable.

To refactor this first code example, I'll start by removing the code that manually updates the index. A good way to do that is with a `for`-loop in Python. Using the `range()` built-in, I can generate the indexes automatically:

```
>>> range(len(my_items))
range(0, 3)
```

```
>>> list(range(0, 3))  
[0, 1, 2]
```

The `range` type represents an immutable sequence of numbers. Its advantage over a regular list is that it always takes the same small amount of memory. Range objects don't actually store the individual values representing the number sequence—instead, they function as iterators and calculate the sequence values on the fly.<sup>1</sup>

So, rather than incrementing `i` manually on each loop iteration, I could take advantage of the `range()` function and write something like this:

```
for i in range(len(my_items)):  
    print(my_items[i])
```

This is better. However, it still isn't very Pythonic and it still feels more like a Java-esque iteration construct than a proper Python loop. When you see code that uses `range(len(...))` to iterate over a container you can usually simplify and improve it further.

As I mentioned, in Python, `for`-loops are really “for-each” loops that can iterate directly over items from a container or sequence, without having to look them up by index. I can use this to simplify this loop even more:

```
for item in my_items:  
    print(item)
```

I would consider this solution to be quite Pythonic. It uses several advanced Python features but remains nice and clean and almost reads like pseudo code from a programming textbook. Notice how this loop

---

<sup>1</sup>In Python 2 you'll need to use the `xrange()` built-in to get this memory-saving behavior, as `range()` will actually construct a list object.

no longer keeps track of the container's size and doesn't use a running index to access elements.

The container itself now takes care of handing out the elements so they can be processed. If the container is ordered, the resulting sequence of elements will be too. If the container isn't ordered, it will return its elements in arbitrary order but the loop will still cover all of them.

Now, of course you won't always be able to rewrite your loops like that. What if you *need* the item index, for example?

It's possible to write loops that keep a running index while avoiding the `range(len(...))` pattern I cautioned against. The `enumerate()` built-in helps you make those kinds of loops nice and Pythonic:

```
>>> for i, item in enumerate(my_items):
...     print(f'{i}: {item}')

0: a
1: b
2: c
```

You see, iterators in Python can return more than just one value. They can return tuples with an arbitrary number of values that can then be unpacked right inside the `for`-statement.

This is very powerful. For example, you can use the same technique to iterate over the keys and values of a dictionary at the same time:

```
>>> emails = {
...     'Bob': 'bob@example.com',
...     'Alice': 'alice@example.com',
... }

>>> for name, email in emails.items():
...     print(f'{name} -> {email}')
```

```
'Bob -> bob@example.com'  
'Alice -> alice@example.com'
```

There's one more example I'd like to show you. What if you absolutely, positively need to write a C-style loop. For example, what if you must control the step size for the index? Imagine you started out with the following Java loop:

```
for (int i = a; i < n; i += s) {  
    // ...  
}
```

How would this pattern translate to Python? The `range()` function comes to our rescue again—it accepts optional parameters to control the start value for the loop (`a`), the stop value (`n`), and the step size (`s`). Therefore, our Java loop example could be translated to Python, like this:

```
for i in range(a, n, s):  
    # ...
```

## Key Takeaways

- Writing C-style loops in Python is considered unpythonic. Avoid managing loop indexes and stop conditions manually if possible.
- Python's for-loops are really “for-each” loops that can iterate directly over items from a container or sequence.

## 6.2 Comprehending Comprehensions

One of my favorite features in Python are list comprehensions. They can seem a bit arcane at first but when you break them down they are actually a very simple construct.

The key to understanding list comprehensions is that they're just for-loops over a collection but expressed in a more terse and compact syntax.

This is sometimes referred to as *syntactic sugar*—a little shortcut for frequently used functionality that makes our lives as Python coders easier. Take the following list comprehension as an example:

```
>>> squares = [x * x for x in range(10)]
```

It computes a list of all integer square numbers from zero to nine:

```
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

If you wanted to build the same list using a plain for-loop, you'd probably write something like this:

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x * x)
```

That's a pretty straightforward loop, right? If you go back and compare the list comprehension example with the for-loop version, you'll spot the commonalities and eventually some patterns will emerge. By generalizing some of the common structure here, you'll eventually end up with a template similar to the one below:



```
values = [expression for item in collection]
```

The above list comprehension “template” is equivalent to the following plain for-loop:

```
values = []
for item in collection:
    values.append(expression)
```

Here, we first set up a new list instance to receive the output values. Then, we iterate over all items in the container, transforming each of them with an arbitrary expression and then adding the individual results to the output list.

This is a “cookie-cutter pattern” that you can apply to many for-loops in order to transform them into list comprehensions and vice versa. Now, there’s one more useful addition we need to make to this template, and that is filtering elements with *conditions*.

List comprehensions can filter values based on some arbitrary condition that decides whether or not the resulting value becomes a part of the output list. Here’s an example:

```
>>> even_squares = [x * x for x in range(10)
                    if x % 2 == 0]
```

This list comprehension will compute a list of the squares of all *even* integers from zero to nine. The *modulo* (%) operator used here returns the remainder after division of one number by another. In this example, we use it to test if a number is even. And it has the desired result:

```
>>> even_squares
[0, 4, 16, 36, 64]
```

Similar to the first example, this new list comprehension can be transformed into an equivalent for-loop:

```
even_squares = []
for x in range(10):
    if x % 2 == 0:
        even_squares.append(x * x)
```

Let's try and generalize the above *list comprehension to for-loop* transformation pattern some more. This time we're going to add a filter condition to our template so we get to decide which values end up in the output list. Here's the updated list comprehension template:

```
values = [expression
          for item in collection
          if condition]
```

Again, we can transform this list comprehension into a *for-loop* with the following pattern:

```
values = []
for item in collection:
    if condition:
        values.append(expression)
```

Once more, this was a straightforward transformation—we simply applied the updated cookie-cutter pattern. I hope this dispels some of the “magic” associated with how list comprehensions work. They're a useful tool that all Python programmers should know how to use.

Before you move on, I want to point out that Python not only supports *list* comprehensions but also has similar syntactic sugar for *sets* and *dictionaries*.

Here's what a *set comprehension* looks like:

```
>>> { x * x for x in range(-9, 10) }  
set([64, 1, 36, 0, 49, 9, 16, 81, 25, 4])
```

Unlike lists, which retain the order of the elements in them, Python sets are an unordered collection type. So you'll get a more or less "random" order when you add items to a set container.

And this is a *dictionary comprehension*:

```
>>> { x: x * x for x in range(5) }  
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Both are useful tools in practice. There's one caveat to Python's comprehensions though—as you get more proficient at using them, it becomes easier and easier to write code that's difficult to read. If you're not careful, you might have to deal with monstrous list, set, and dict comprehensions soon. Remember, too much of a good thing is usually a bad thing.

After much chagrin, I'm personally drawing the line at one level of nesting for comprehensions. I found that in most cases it's better (as in "more readable" and "easier to maintain") to use for-loops beyond that point.

## Key Takeaways

- Comprehensions are a key feature in Python. Understanding and applying them will make your code much more Pythonic.
- Comprehensions are just fancy syntactic sugar for a simple for-loop pattern. Once you understand the pattern, you'll develop an intuitive understanding for comprehensions.
- There are more than just list comprehensions.

## 6.3 List Slicing Tricks and the Sushi Operator

Python’s list objects have a neat feature called *slicing*. You can view it as an extension of the square-brackets indexing syntax. Slicing is commonly used to access ranges of elements within an ordered collection. For example, you can slice up a large list object into several smaller sublists with it.

Here’s an example. Slicing uses the familiar “[ ]” indexing syntax with the following “[start:stop:step]” pattern:

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst
[1, 2, 3, 4, 5]

#   lst[start:end:step]
>>> lst[1:3:1]
[2, 3]
```

Adding the [1:3:1] index returned a slice of the original list ranging from index 1 to index 2, with a step size of one element. To avoid off-by-one errors, it’s important to remember that the upper bound is always exclusive. This is why we got [2, 3] as the sublist from the [1:3:1] slice.

If you leave out the step size, it defaults to one:

```
>>> lst[1:3]
[2, 3]
```

You can do other interesting things with the step parameter, also called the *stride*. For example, you can create a sublist that includes every other element of the original:

```
>>> lst[:2]
[1, 3, 5]
```

Wasn't that fun? I like to call ":" the *sushi operator*. It looks like a delicious maki roll cut in half. Besides reminding you of delicious food and accessing ranges of lists, it has a few more lesser-known applications. Let me show you some more fun and useful list-slicing tricks!

You just saw how the slicing step size can be used to select every other element of a list. Well, there's more: If you ask for a `[:-1]` slice, you'll get a copy of the original list, but in the reverse order:

```
>>> numbers[::-1]
[5, 4, 3, 2, 1]
```

We asked Python to give us the full list (`:`), but to go over all of the elements from back to front by setting the step size to `-1`. This is pretty neat, but in most cases I'd still stick with `list.reverse()` and the built-in `reversed()` function to reverse a list.

Here's another list-slicing trick: You can use the `:-` operator to clear all elements from a list without destroying the list object itself.

This is extremely helpful when you need to clear out a list in your program that has other references pointing to it. In this case, you often can't just empty the list by replacing it with a new list object, since that wouldn't update the other references. But here's the sushi operator coming to your rescue:

```
>>> lst = [1, 2, 3, 4, 5]
>>> del lst[:]
>>> lst
[]
```

As you can see, this removes all elements from `lst` but leaves the list object itself intact. In Python 3 you can also use `lst.clear()` for the

same job, which might be the more readable pattern, depending on the circumstances. However, keep in mind that `clear()` isn't available in Python 2.

Besides clearing lists, you can also use slicing to replace all elements of a list without creating a new list object. This is a nice shorthand for clearing a list and then repopulating it manually:

```
>>> original_lst = lst
>>> lst[:] = [7, 8, 9]
>>> lst
[7, 8, 9]
>>> original_lst
[7, 8, 9]
>>> original_lst is lst
True
```

The previous code example replaced all elements in `lst` but did not destroy and recreate the list itself. The old references to the original list object are therefore still valid.

Yet another use case for the sushi operator is creating (shallow) copies of existing lists:

```
>>> copied_lst = lst[:]
>>> copied_lst
[7, 8, 9]
>>> copied_lst is lst
False
```

Creating a *shallow* copy means that only the structure of the elements is copied, not the elements themselves. Both copies of the list share the same instances of the individual elements.

If you need to duplicate everything, including the elements, then you'll need to create a *deep* copy of the list. Python's built-in `copy` module will come in handy for this.

## Key Takeaways

- The `:` “sushi operator” is not only useful for selecting sublists of elements within a list. It can also be used to clear, reverse, and copy lists.
- But be careful—this functionality borders on the arcane for many Python developers. Using it might make your code less maintainable for everyone else on your team.

## 6.4 Beautiful Iterators

I love how beautiful and clear Python’s syntax is compared to many other programming languages. Let’s take the humble *for-in* loop, for example. It speaks to Python’s beauty that you can read a Pythonic loop like this, as if it was an English sentence:

```
numbers = [1, 2, 3]
for n in numbers:
    print(n)
```

But how do Python’s elegant loop constructs work behind the scenes? How does the loop fetch individual elements from the object it is looping over? And, how can you support the same programming style in your own Python objects?

You’ll find the answers to these questions in Python’s *iterator protocol*: Objects that support the `__iter__` and `__next__` dunder methods automatically work with *for-in* loops.

But let’s take things step by step. Just like decorators, iterators and their related techniques can appear quite arcane and complicated on first glance. So, we’ll ease into them.

In this chapter you’ll see how to write several Python classes that support the iterator protocol. They’ll serve as “non-magical” examples and test implementations you can build upon and deepen your understanding with.

We’ll focus on the core mechanics of iterators in Python 3 first and leave out any unnecessary complications, so you can see clearly how iterators behave at the fundamental level.

I’ll tie each example back to the *for-in* loop question we started out with. And, at the end of this chapter we’ll go over some differences that exist between Python 2 and 3 when it comes to iterators.

Ready? Let’s jump right in!



## Iterating Forever

We'll begin by writing a class that demonstrates the bare-bones iterator protocol. The example I'm using here might look different from the examples you've seen in other iterator tutorials, but bear with me. I think doing it this way gives you a more applicable understanding of how iterators work in Python.

Over the next few paragraphs we're going to implement a class called `Repeater` that can be iterated over with a *for-in* loop, like so:

```
repeater = Repeater('Hello')
for item in repeater:
    print(item)
```

Like its name suggests, instances of this `Repeater` class will repeatedly return a single value when iterated over. So the above example code would forever print the string `'Hello'` to the console.

To start with the implementation, we'll first define and flesh out the `Repeater` class:

```
class Repeater:
    def __init__(self, value):
        self.value = value

    def __iter__(self):
        return RepeaterIterator(self)
```

On first inspection, `Repeater` looks like a bog-standard Python class. But notice how it also includes the `__iter__` dunder method.

What's the `RepeaterIterator` object we're creating and returning from `__iter__`? It's a helper class we also need to define for our *for-in* iteration example to work:

```
class RepeaterIterator:
    def __init__(self, source):
        self.source = source

    def __next__(self):
        return self.source.value
```

Again, `RepeaterIterator` looks like a straightforward Python class, but you might want to take note of the following two things:

1. In the `__init__` method, we link each `RepeaterIterator` instance to the `Repeater` object that created it. That way we can hold onto the “source” object that’s being iterated over.
2. In `RepeaterIterator.__next__`, we reach back into the “source” `Repeater` instance and return the value associated with it.

In this code example, `Repeater` and `RepeaterIterator` are working *together* to support Python’s iterator protocol. The two dunder methods we defined, `__iter__` and `__next__`, are the keys to making a Python object iterable.

We’ll take a closer look at these two methods and how they work together after some hands-on experimentation with the code we’ve got so far.

Let’s confirm that this two-class setup really made `Repeater` objects compatible with *for-in* loop iteration. To do that we’ll first create an instance of `Repeater` that would return the string ‘Hello’ indefinitely:

```
>>> repeater = Repeater('Hello')
```

And now we’re going to try iterating over this `repeater` object with a *for-in* loop. What’s going to happen when you run the following code snippet?

```
>>> for item in repeater:  
...     print(item)
```

Right on! You'll see 'Hello' printed to the screen...a lot. Repeater keeps on returning the same string value, and so, this loop will never complete. Our little program is doomed to forever print 'Hello' to the console:

```
Hello  
Hello  
Hello  
Hello  
Hello  
...
```

But congratulations—you just wrote a working iterator in Python and used it with a *for-in* loop. The loop may not terminate yet...but so far, so good!

Next up, we'll tease this example apart to understand how the `__iter__` and `__next__` methods work together to make a Python object iterable.

Pro tip: If you ran the last example inside a Python REPL session or from the terminal, and you want to stop it, hit *Ctrl* + *C* a few times to break out of the infinite loop.

## How do *for-in* loops work in Python?

At this point we've got our Repeater class that apparently supports the iterator protocol, and we just ran a *for-in* loop to prove it:

```
repeater = Repeater('Hello')  
for item in repeater:  
    print(item)
```

Now, what does this `for-in` loop really do behind the scenes? How does it communicate with the `repeater` object to fetch new elements from it?

To dispel some of that “magic,” we can expand this loop into a slightly longer code snippet that gives the same result:

```
repeater = Repeater('Hello')
iterator = repeater.__iter__()
while True:
    item = iterator.__next__()
    print(item)
```

As you can see, the *for-in* was just syntactic sugar for a simple `while` loop:

- It first prepared the `repeater` object for iteration by calling its `__iter__` method. This returned the actual *iterator object*.
- After that, the loop repeatedly called the iterator object’s `__next__` method to retrieve values from it.

If you’ve ever worked with *database cursors*, this mental model will seem familiar: We first initialize the cursor and prepare it for reading, and then we can fetch data from it into local variables as needed, one element at a time.

Because there’s never more than one element “in flight,” this approach is highly memory-efficient. Our `Repeater` class provides an *infinite* sequence of elements and we can iterate over it just fine. Emulating the same thing with a Python list would be impossible—there’s no way we could create a list with an infinite number of elements in the first place. This makes iterators a very powerful concept.

On more abstract terms, iterators provide a common interface that allows you to process every element of a container while being completely isolated from the container’s internal structure.

Whether you’re dealing with a list of elements, a dictionary, an infinite sequence like the one provided by our Repeater class, or another sequence type—all of that is just an implementation detail. Every single one of these objects can be traversed in the same way with the power of iterators.

And as you’ve seen, there’s nothing special about *for-in* loops in Python. If you peek behind the curtain, it all comes down to calling the right dunder methods at the right time.

In fact, you can manually “emulate” how the loop uses the iterator protocol in a Python interpreter session:

```
>>> repeater = Repeater('Hello')
>>> iterator = iter(repeater)
>>> next(iterator)
'Hello'
>>> next(iterator)
'Hello'
>>> next(iterator)
'Hello'
...

```

This gives the same result—an infinite stream of hellos. Every time you call `next()`, the iterator hands out the same greeting again.

By the way, I took the opportunity here to replace the calls to `__iter__` and `__next__` with calls to Python’s built-in functions, `iter()` and `next()`.

Internally, these built-ins invoke the same dunder methods, but they make this code a little prettier and easier to read by providing a clean “facade” to the iterator protocol.

Python offers these facades for other functionality as well. For example, `len(x)` is a shortcut for calling `x.__len__`. Similarly, calling `iter(x)` invokes `x.__iter__` and calling `next(x)` invokes `x.__next__`.

Generally, it's a good idea to use the built-in facade functions rather than directly accessing the dunder methods implementing a protocol. It just makes the code a little easier to read.

## A Simpler Iterator Class

Up until now, our iterator example consisted of two separate classes, `Repeater` and `RepeaterIterator`. They corresponded directly to the two phases used by Python's iterator protocol:

First, setting up and retrieving the iterator object with an `iter()` call, and then repeatedly fetching values from it via `next()`.

Many times *both of these responsibilities* can be shouldered by a single class. Doing this allows you to reduce the amount of code necessary to write a class-based iterator.

I chose not to do this with the first example in this chapter because it mixes up the cleanliness of the mental model behind the iterator protocol. But now that you've seen how to write a class-based iterator the longer and more complicated way, let's take a minute to simplify what we've got so far.

Remember why we needed the `RepeaterIterator` class again? We needed it to host the `__next__` method for fetching new values from the iterator. But it doesn't really matter *where* `__next__` is defined. In the iterator protocol, all that matters is that `__iter__` returns *any* object with a `__next__` method on it.

So here's an idea: `RepeaterIterator` returns the same value over and over, and it doesn't have to keep track of any internal state. What if we added the `__next__` method directly to the `Repeater` class instead?

That way we could get rid of `RepeaterIterator` altogether and implement an iterable object with a single Python class. Let's try it out! Our new and simplified iterator example looks as follows:

```
class Repeater:
    def __init__(self, value):
        self.value = value

    def __iter__(self):
        return self

    def __next__(self):
        return self.value
```

We just went from two separate classes and 10 lines of code to just one class and 7 lines of code. Our simplified implementation still supports the iterator protocol just fine:

```
>>> repeater = Repeater('Hello')
>>> for item in repeater:
...     print(item)

Hello
Hello
Hello
...
```

Streamlining a class-based iterator like that often makes sense. In fact, most Python iterator tutorials start out that way. But I always felt that explaining iterators with a single class from the get-go hides the underlying principles of the iterator protocol—and thus makes it more difficult to understand.

## Who Wants to Iterate Forever

At this point you should have a pretty good understanding of how iterators work in Python. But so far we've only implemented iterators that keep on iterating *forever*.

Clearly, infinite repetition isn't the main use case for iterators in Python. In fact, when you look back all the way to the beginning of this chapter, I used the following snippet as a motivating example:

```
numbers = [1, 2, 3]
for n in numbers:
    print(n)
```

You'll rightfully expect this code to print the numbers 1, 2, and 3 and then stop. And you probably *wouldn't* expect it to go on spamming your terminal window by printing "3" forever until you mash *Ctrl+C* a few times in a wild panic...

And so, it's time to find out how to write an iterator that eventually *stops* generating new values instead of iterating forever because that's what Python objects typically do when we use them in a *for-in* loop.

We'll now write another iterator class that we'll call `BoundedRepeater`. It'll be similar to our previous `Repeater` example, but this time we'll want it to stop after a predefined number of repetitions.

Let's think about this for a bit. How do we do this? How does an iterator signal that it's exhausted and out of elements to iterate over? Maybe you're thinking, "Hmm, we could just return `None` from the `__next__` method."

And that's not a bad idea—but the trouble is, what are we going to do if we *want* some iterators to be able to return `None` as an acceptable value?

Let's see what other Python iterators do to solve this problem. I'm going to construct a simple container, a list with a few elements, and then I'll iterate over it until it runs out of elements to see what happens:

```
>>> my_list = [1, 2, 3]
>>> iterator = iter(my_list)
```



```
>>> next(iterator)
1
>>> next(iterator)
2
>>> next(iterator)
3
```

Careful now! We’ve consumed all of the three available elements in the list. Watch what happens if I call `next` on the iterator again:

```
>>> next(iterator)
StopIteration
```

Aha! It raises a `StopIteration` exception to signal we’ve exhausted all of the available values in the iterator.

That’s right: Iterators use exceptions to structure control flow. To signal the end of iteration, a Python iterator simply raises the built-in `StopIteration` exception.

If I keep requesting more values from the iterator, it’ll keep raising `StopIteration` exceptions to signal that there are no more values available to iterate over:

```
>>> next(iterator)
StopIteration
>>> next(iterator)
StopIteration
...
```

Python iterators normally can’t be “reset”—once they’re exhausted they’re supposed to raise `StopIteration` every time `next()` is called on them. To iterate anew you’ll need to request a fresh iterator object with the `iter()` function.

Now we know everything we need to write our `BoundedRepeater` class that stops iterating after a set number of repetitions:

```
class BoundedRepeater:
    def __init__(self, value, max_repeats):
        self.value = value
        self.max_repeats = max_repeats
        self.count = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.count >= self.max_repeats:
            raise StopIteration
        self.count += 1
        return self.value
```

This gives us the desired result. Iteration stops after the number of repetitions defined in the `max_repeats` parameter:

```
>>> repeater = BoundedRepeater('Hello', 3)
>>> for item in repeater:
    print(item)

Hello
Hello
Hello
```

If we rewrite this last for-in loop example to take away some of the syntactic sugar, we end up with the following expanded code snippet:

```
repeater = BoundedRepeater('Hello', 3)
iterator = iter(repeater)
while True:
    try:
```

```
        item = next(iterator)
    except StopIteration:
        break
    print(item)
```

Every time `next()` is called in this loop, we check for a `StopIteration` exception and break the while loop if necessary.

Being able to write a three-line *for-in* loop instead of an eight-line while loop is quite a nice improvement. It makes the code easier to read and more maintainable. And this is another reason why iterators in Python are such a powerful tool.

## Python 2.x Compatibility

All the code examples I showed here were written in Python 3. There's a small but important difference between Python 2 and 3 when it comes to implementing class-based iterators:

- In Python 3, the method that retrieves the next value from an iterator is called `__next__`.
- In Python 2, the same method is called `next` (no underscores).

This naming difference can lead to some trouble if you're trying to write class-based iterators that should work on both versions of Python. Luckily, there's a simple approach you can take to work around this difference.

Here's an updated version of the `InfiniteRepeater` class that will work on both Python 2 and Python 3:

```
class InfiniteRepeater(object):
    def __init__(self, value):
        self.value = value
```

```
def __iter__(self):  
    return self  
  
def __next__(self):  
    return self.value  
  
# Python 2 compatibility:  
def next(self):  
    return self.__next__()
```

To make this iterator class compatible with Python 2, I've made two small changes to it:

First, I added a `next` method that simply calls the original `__next__` and forwards its return value. This essentially creates an alias for the existing `__next__` implementation so that Python 2 finds it. That way we can support both versions of Python while still keeping all of the actual implementation details in one place.

And second, I modified the class definition to inherit from `object` in order to ensure we're creating a *new-style* class on Python 2. This has nothing to do with iterators specifically, but it's a good practice nonetheless.

## Key Takeaways

- Iterators provide a sequence interface to Python objects that's memory efficient and considered Pythonic. Behold the beauty of the *for-in* loop!
- To support iteration an object needs to implement the *iterator protocol* by providing the `__iter__` and `__next__` dunder methods.
- Class-based iterators are only one way to write iterable objects in Python. Also consider generators and generator expressions.

## 6.5 Generators Are Simplified Iterators

In the chapter on iterators we spent quite a bit of time writing a class-based iterator. This wasn't a bad idea from an educational perspective—but it also demonstrated how writing an iterator class requires a lot of boilerplate code. To tell you the truth, as a “lazy” developer, I don't like tedious and repetitive work.

And yet, iterators are so useful in Python. They allow you to write pretty *for-in* loops and help you make your code more Pythonic and efficient. If there only was a more convenient way to write these iterators in the first place...

Surprise, there is! Once more, Python helps us out with some syntactic sugar to make writing iterators easier. In this chapter you'll see how to write iterators faster and with less code using *generators* and the `yield` keyword.

### Infinite Generators

Let's start by looking again at the Repeater example that I previously used to introduce the idea of iterators. It implemented a class-based iterator cycling through an infinite sequence of values. This is what the class looked like in its second (simplified) version:

```
class Repeater:
    def __init__(self, value):
        self.value = value

    def __iter__(self):
        return self

    def __next__(self):
        return self.value
```

If you're thinking, “that's quite a lot of code for such a simple iterator,” you're absolutely right. Parts of this class seem rather formulaic, as if

they would be written in exactly the same way from one class-based iterator to the next.

This is where Python's *generators* enter the scene. If I rewrite this iterator class as a generator, it looks like this:

```
def repeater(value):  
    while True:  
        yield value
```

We just went from seven lines of code to three. Not bad, eh? As you can see, generators look like regular functions but instead of using the return statement, they use `yield` to pass data back to the caller.

Will this new generator implementation still work the same way as our class-based iterator did? Let's bust out the *for-in* loop test to find out:

```
>>> for x in repeater('Hi'):  
...     print(x)  
'Hi '  
'Hi '  
'Hi '  
'Hi '  
'Hi '  
...
```

Yep! We're still looping through our greetings forever. This much shorter *generator* implementation seems to perform the same way that the *Repeater* class did. (Remember to hit *Ctrl+C* if you want out of the infinite loop in an interpreter session.)

Now, how do these generators work? They look like normal functions, but their behavior is quite different. For starters, calling a generator function doesn't even run the function. It merely creates and returns a *generator object*:

```
>>> repeater('Hey')
<generator object repeater at 0x107bcdbf8>
```

The code in the generator function only executes when `next()` is called on the generator object:

```
>>> generator_obj = repeater('Hey')
>>> next(generator_obj)
'Hey'
```

If you read the code of the `repeater` function again, it looks like the `yield` keyword in there somehow stops this generator function in mid-execution and then resumes it at a later point in time:

```
def repeater(value):
    while True:
        yield value
```

And that's quite a fitting mental model for what happens here. You see, when a `return` statement is invoked inside a function, it permanently passes control back to the caller of the function. When a `yield` is invoked, it also passes control back to the caller of the function—but it only does so *temporarily*.

Whereas a `return` statement disposes of a function's local state, a `yield` statement suspends the function and retains its local state. In practical terms, this means local variables and the execution state of the generator function are only stashed away temporarily and not thrown out completely. Execution can be resumed at any time by calling `next()` on the generator:

```
>>> iterator = repeater('Hi')
>>> next(iterator)
'Hi'
```

```
>>> next(iterator)
'Hi '
>>> next(iterator)
'Hi '
```

This makes generators fully compatible with the iterator protocol. For this reason, I like to think of them primarily as syntactic sugar for implementing iterators.

You'll find that for most types of iterators, writing a generator function will be easier and more readable than defining a long-winded class-based iterator.

## Generators That Stop Generating

In this chapter we started out by writing an *infinite* generator once again. By now you're probably wondering how to write a generator that stops producing values after a while, instead of going on and on forever.

Remember, in our class-based iterator we were able to signal the end of iteration by manually raising a `StopIteration` exception. Because generators are fully compatible with class-based iterators, that's still what happens behind the scenes.

Thankfully, as programmers we get to work with a nicer interface this time around. Generators stop generating values as soon as control flow returns from the generator function by any means other than a `yield` statement. This means you no longer have to worry about raising `StopIteration` at all!

Here's an example:

```
def repeat_three_times(value):
    yield value
    yield value
    yield value
```



Notice how this generator function doesn't include any kind of loop. In fact it's dead simple and only consists of three `yield` statements. If a `yield` temporarily suspends execution of the function and passes back a value to the caller, what will happen when we reach the end of this generator? Let's find out:

```
>>> for x in repeat_three_times('Hey there'):
...     print(x)
'Hey there'
'Hey there'
'Hey there'
```

As you may have expected, this generator stopped producing new values after three iterations. We can assume that it did so by raising a `StopIteration` exception when execution reached the end of the function. But to be sure, let's confirm that with another experiment:

```
>>> iterator = repeat_three_times('Hey there')
>>> next(iterator)
'Hey there'
>>> next(iterator)
'Hey there'
>>> next(iterator)
'Hey there'
>>> next(iterator)
StopIteration
>>> next(iterator)
StopIteration
```

This iterator behaved just like we expected. As soon as we reach the end of the generator function, it keeps raising `StopIteration` to signal that it has no more values to provide.

Let's come back to another example from the iterators chapter. The `BoundedIterator` class implemented an iterator that would only repeat a value a set number of times:

```
class BoundedRepeater:
    def __init__(self, value, max_repeats):
        self.value = value
        self.max_repeats = max_repeats
        self.count = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.count >= self.max_repeats:
            raise StopIteration
        self.count += 1
        return self.value
```

Why don't we try to re-implement this BoundedRepeater class as a generator function. Here's my first take on it:

```
def bounded_repeater(value, max_repeats):
    count = 0
    while True:
        if count >= max_repeats:
            return
        count += 1
        yield value
```

I intentionally made the while loop in this function a little unwieldy. I wanted to demonstrate how invoking a return statement from a generator causes iteration to stop with a StopIteration exception. We'll soon clean up and simplify this generator function some more, but first let's try out what we've got so far:

```
>>> for x in bounded_repeater('Hi', 4):
...     print(x)
'Hi'
```

```
'Hi '  
'Hi '  
'Hi '
```

Great! Now we have a generator that stops producing values after a configurable number of repetitions. It uses the `yield` statement to pass back values until it finally hits the `return` statement and iteration stops.

Like I promised you, we can further simplify this generator. We'll take advantage of the fact that Python adds an implicit `return None` statement to the end of every function. This is what our final implementation looks like:

```
def bounded_repeater(value, max_repeats):  
    for i in range(max_repeats):  
        yield value
```

Feel free to confirm that this simplified generator still works the same way. All things considered, we went from a 12-line implementation in the `BoundedRepeater` class to a three-line generator-based implementation providing the exact same functionality. That's a 75% reduction in the number of lines of code—not too shabby!

As you just saw, generators help “abstract away” most of the boilerplate code otherwise needed when writing class-based iterators. They can make your life as a programmer much easier and allow you to write cleaner, shorter, and more maintainable iterators. Generator functions are a great feature in Python, and you shouldn't hesitate to use them in your own programs.

## Key Takeaways

- Generator functions are syntactic sugar for writing objects that support the iterator protocol. Generators abstract away much

of the boilerplate code needed when writing class-based iterators.

- The `yield` statement allows you to temporarily suspend execution of a generator function and to pass back values from it.
- Generators start raising `StopIteration` exceptions after control flow leaves the generator function by any means other than a `yield` statement.

## 6.6 Generator Expressions

As I learned more about Python’s iterator protocol and the different ways to implement it in my own code, I realized that “syntactic sugar” was a recurring theme.

You see, class-based iterators and generator functions are two expressions of the same underlying design pattern.

Generator functions give you a shortcut for supporting the iterator protocol in your own code, and they avoid much of the verbosity of class-based iterators. With a little bit of specialized syntax, or *syntactic sugar*, they save you time and make your life as a developer easier.

This is a recurring theme in Python and in other programming languages. As more developers use a design pattern in their programs, there’s a growing incentive for the language creators to provide abstractions and implementation shortcuts for it.

That’s how programming languages evolve over time—and as developers, we reap the benefits. We get to work with more and more powerful building blocks, which reduces busywork and lets us achieve more in less time.

Earlier in this book you saw how generators provide syntactic sugar for writing class-based iterators. The *generator expressions* we’ll cover in this chapter add another layer of syntactic sugar on top.

Generator expressions give you an even more effective shortcut for writing iterators. With a simple and concise syntax that looks like a list comprehension, you’ll be able to define iterators in a single line of code.

Here’s an example:

```
iterator = ('Hello' for i in range(3))
```

When iterated over, this generator expression yields the same

sequence of values as the `bounded_repeater` generator function we wrote in the previous chapter. Here it is again to refresh your memory:

```
def bounded_repeater(value, max_repeats):  
    for i in range(max_repeats):  
        yield value  
  
iterator = bounded_repeater('Hello', 3)
```

Isn't it amazing how a single-line generator expression now does a job that previously required a four-line generator function or a much longer class-based iterator?

But I'm getting ahead of myself. Let's make sure our iterator defined with a generator expression actually works as expected:

```
>>> iterator = ('Hello' for i in range(3))  
>>> for x in iterator:  
...     print(x)  
'Hello'  
'Hello'  
'Hello'
```

That looks pretty good to me! We seem to get the same results from our one-line generator expression that we got from the `bounded_repeater` generator function.

There's one small caveat though: Once a generator expression has been consumed, it can't be restarted or reused. So in some cases there is an advantage to using generator functions or class-based iterators.

## Generator Expressions vs List Comprehensions

As you can tell, generator expressions are somewhat similar to list comprehensions:

```
>>> listcomp = ['Hello' for i in range(3)]
>>> genexpr = ('Hello' for i in range(3))
```

Unlike list comprehensions, however, generator expressions don't construct list objects. Instead, they generate values “just in time” like a class-based iterator or generator function would.

All you get by assigning a generator expression to a variable is an iterable “generator object”:

```
>>> listcomp
['Hello', 'Hello', 'Hello']

>>> genexpr
<generator object <genexpr> at 0x1036c3200>
```

To access the values produced by the generator expression, you need to call `next()` on it, just like you would with any other iterator:

```
>>> next(genexpr)
'Hello'
>>> next(genexpr)
'Hello'
>>> next(genexpr)
'Hello'
>>> next(genexpr)
StopIteration
```

Alternatively, you can also call the `list()` function on a generator expression to construct a list object holding all generated values:

```
>>> genexpr = ('Hello' for i in range(3))
>>> list(genexpr)
['Hello', 'Hello', 'Hello']
```

Of course, this was just a toy example to show how you can “convert” a generator expression (or any other iterator for that matter) into a list. If you need a list object right away, you’d normally just write a list comprehension from the get-go.

Let’s take a closer look at the syntactic structure of this simple generator expression. The pattern you should begin to see looks like this:

```
genexpr = (expression for item in collection)
```

The above generator expression “template” corresponds to the following generator function:

```
def generator():  
    for item in collection:  
        yield expression
```

Just like with list comprehensions, this gives you a “cookie-cutter pattern” you can apply to many generator functions in order to transform them into concise *generator expressions*.

## Filtering Values

There’s one more useful addition we can make to this template, and that’s element filtering with conditions. Here’s an example:

```
>>> even_squares = (x * x for x in range(10)  
                    if x % 2 == 0)
```

This generator yields the square numbers of all even integers from zero to nine. The filtering condition using the % (modulo) operator will reject any value not divisible by two:



```
>>> for x in even_squares:
...     print(x)
0
4
16
36
64
```

Let's update our generator expression template. After adding element filtering via if-conditions, the template now looks like this:

```
genexpr = (expression for item in collection
             if condition)
```

And once again, this pattern corresponds to a relatively straightforward, but longer, generator function. Syntactic sugar at its best:

```
def generator():
    for item in collection:
        if condition:
            yield expression
```

## In-line Generator Expressions

Because generator expressions are, well...expressions, you can use them in-line with other statements. For example, you can define an iterator and consume it right away with a for-loop:

```
for x in ('Bom dia' for i in range(3)):
    print(x)
```

There's another syntactic trick you can use to make your generator expressions more beautiful. The parentheses surrounding a generator expression can be dropped if the generator expression is used as the single argument to a function:

```
>>> sum((x * 2 for x in range(10)))  
90
```

*# Versus:*

```
>>> sum(x * 2 for x in range(10))  
90
```

This allows you to write concise and performant code. Because generator expressions generate values “just in time” like a class-based iterator or a generator function would, they are very memory efficient.

## Too Much of a Good Thing...

Like list comprehensions, generator expressions allow for more complexity than what we’ve covered so far. Through nested for-loops and chained filtering clauses, they can cover a wider range of use cases:

```
(expr for x in xs if cond1  
      for y in ys if cond2  
      ...  
      for z in zs if condN)
```

The above pattern translates to the following generator function logic:

```
for x in xs:  
    if cond1:  
        for y in ys:  
            if cond2:  
                ...  
                for z in zs:  
                    if condN:  
                        yield expr
```

And this is where I’d like to place a big caveat:

Please don't write deeply nested generator expressions like that. They can be very difficult to maintain in the long run.

This is one of those “the dose makes the poison” situations where a beautiful and simple tool can be overused to create hard to read and difficult to debug programs.

Just like with list comprehensions, I personally try to stay away from any generator expression that includes more than two levels of nesting.

Generator expressions are a helpful and Pythonic tool in your toolbox, but that doesn't mean they should be used for every single problem you're facing. For complex iterators, it's often better to write a generator function or even a class-based iterator.

If you need to use nested generators and complex filtering conditions, it's usually better to factor out sub-generators (so you can name them) and then to chain them together again at the top level. You'll see how to do this in the next chapter on *iterator chains*.

If you're on the fence, try out different implementations and then select the one that seems the most readable. Trust me, it'll save you time in the long run.

### Key Takeaways

- Generator expressions are similar to list comprehensions. However, they don't construct list objects. Instead, generator expressions generate values “just in time” like a class-based iterator or generator function would.
- Once a generator expression has been consumed, it can't be restarted or reused.
- Generator expressions are best for implementing simple “ad hoc” iterators. For complex iterators, it's better to write a generator function or a class-based iterator.

## 6.7 Iterator Chains

Here's another great feature of iterators in Python: By chaining together multiple iterators you can write highly efficient data processing “pipelines.” The first time I saw this pattern in action in a PyCon presentation by David Beazley, it blew my mind.

If you take advantage of Python's generator functions and generator expressions, you'll be building concise and powerful *iterator chains* in no time. In this chapter you'll find out what this technique looks like in practice and how you can use it in your own programs.

As a quick recap, generators and generator expressions are syntactic sugar for writing iterators in Python. They abstract away much of the boilerplate code needed when writing class-based iterators.

While a regular function produces a single return value, generators produce a sequence of results. You could say they *generate a stream of values* over the course of their lifetime.

For example, I can define the following generator that produces the series of integer values from one to eight by keeping a running counter and yielding a new value every time `next()` gets called on it:

```
def integers():  
    for i in range(1, 9):  
        yield i
```

You can confirm this behaviour by running the following code in a Python REPL:

```
>>> chain = integers()  
>>> list(chain)  
[1, 2, 3, 4, 5, 6, 7, 8]
```

So far, so not-very-interesting. But we'll quickly change this now. You

see, generators can be “connected” to each other in order to build efficient data processing algorithms that work like a pipeline.

You can take the “stream” of values coming out of the `integers()` generator and feed them into another generator again. For example, one that takes each number, squares it, and then passes it on:

```
def squared(seq):  
    for i in seq:  
        yield i * i
```

This is what our “data pipeline” or “chain of generators” would do now:

```
>>> chain = squared(integers())  
>>> list(chain)  
[1, 4, 9, 16, 25, 36, 49, 64]
```

And we can keep on adding new building blocks to this pipeline. Data flows in one direction only, and each processing step is shielded from the others via a well-defined interface.

This is similar to how pipelines work in Unix. We chain together a sequence of processes so that the output of each process feeds directly as input to the next one.

Why don’t we add another step to our pipeline that negates each value and then passes it on to the next processing step in the chain:

```
def negated(seq):  
    for i in seq:  
        yield -i
```

If we rebuild our chain of generators and add `negated` at the end, this is the output we get now:

```
>>> chain = negated(squared(integers()))
>>> list(chain)
[-1, -4, -9, -16, -25, -36, -49, -64]
```

My favorite thing about chaining generators is that the data processing happens *one element at a time*. There’s no buffering between the processing steps in the chain:

1. The `integers` generator yields a single value, let’s say 3.
2. This “activates” the `squared` generator, which processes the value and passes it on to the next stage as  $3 \times 3 = 9$
3. The square number yielded by the `squared` generator gets fed immediately into the `negated` generator, which modifies it to -9 and yields it again.

You could keep extending this chain of generators to build out a processing pipeline with many steps. It would still perform efficiently and could easily be modified because each step in the chain is an individual generator function.

Each individual generator function in this processing pipeline is quite concise. With a little trick, we can shrink down the definition of this pipeline even more, without sacrificing much readability:

```
integers = range(8)
squared = (i * i for i in integers)
negated = (-i for i in squared)
```

Notice how I’ve replaced each processing step in the chain with a *generator expression* built on the output of the previous step. This code is equivalent to the chain of generators we built throughout the chapter:

```
>>> negated
<generator object <genexpr> at 0x1098bcb48>
>>> list(negated)
[0, -1, -4, -9, -16, -25, -36, -49]
```

The only downside to using generator expressions is that they can't be configured with function arguments, and you can't reuse the same generator expression multiple times in the same processing pipeline.

But of course, you could mix-and-match generator expressions and regular generators freely in building these pipelines. This will help improve readability with complex pipelines.

## Key Takeaways

- Generators can be chained together to form highly efficient and maintainable data processing pipelines.
- Chained generators process each element going through the chain individually.
- Generator expressions can be used to write concise pipeline definitions, but this can impact readability.

## **Chapter 7**

# **Dictionary Tricks**



## 7.1 Dictionary Default Values

Python's dictionaries have a `get()` method for looking up a key while providing a fallback value. This can be handy in many situations. Let me give you a simple example to show you what I mean. Imagine we have the following data structure that's mapping user IDs to user names:

```
name_for_userid = {
    382: 'Alice',
    950: 'Bob',
    590: 'Dilbert',
}
```

Now we'd like to use this data structure to write a function `greeting()` which will return a greeting for a user based on their user ID. Our first implementation might look something like this:

```
def greeting(userid):
    return 'Hi %s!' % name_for_userid[userid]
```

It's a straightforward dictionary lookup. This first implementation technically works—but only if the user ID is a valid key in the `name_for_userid` dictionary. If we pass an *invalid* user ID to our `greeting` function it throws an exception:

```
>>> greeting(382)
'Hi Alice!'

>>> greeting(33333333)
KeyError: 33333333
```

A `KeyError` exception isn't really the result we'd like to see. It would be much nicer if the function returned a generic greeting as a fallback if the user ID can't be found.

Let's implement this idea. Our first approach might be to simply do a *key in dict* membership check and to return a default greeting if the user ID is unknown:

```
def greeting(userid):  
    if userid in name_for_userid:  
        return 'Hi %s!' % name_for_userid[userid]  
    else:  
        return 'Hi there!'
```

Let's see how this implementation of `greeting()` fares with our previous test cases:

```
>>> greeting(382)  
'Hi Alice!'  
  
>>> greeting(33333333)  
'Hi there!'
```

Much better. We now get a generic greeting for unknown users and we keep the personalized greeting when a valid user ID is found.

But there's still room for improvement. While this new implementation gives us the expected results and seems small and clean enough, it can still be improved. I've got some gripes with the current approach:

- It's *inefficient* because it queries the dictionary twice.
- It's *verbose* since part of the greeting string is repeated, for example.
- It's not *Pythonic*—the official Python documentation specifically recommends an “easier to ask for forgiveness than permission” (EAFP) coding style for these situations:

“This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false.”<sup>1</sup>

A better implementation that follows the *EAFP* principle might use a *try...except* block to catch the `KeyError` instead of doing an explicit membership test:

```
def greeting(userid):  
    try:  
        return 'Hi %s!' % name_for_userid[userid]  
    except KeyError:  
        return 'Hi there'
```

This implementation is still correct as far as our initial requirements go, and now we’ve removed the need for querying the dictionary twice.

But we can still improve this further and come up with a cleaner solution. Python’s dictionaries have a `get()` method on them which supports a “default” parameter that can be used as a fallback value:<sup>2</sup>

```
def greeting(userid):  
    return 'Hi %s!' % name_for_userid.get(  
        userid, 'there')
```

When `get()` is called, it checks if the given key exists in the dictionary. If it does, the value for the key is returned. If it does *not* exist, then the value of the default parameter is returned instead. As you can see, this implementation of `greeting` still works as intended:

```
>>> greeting(950)  
'Hi Bob!'
```

---

<sup>1</sup>cf. Python Glossary: “EAFP”

<sup>2</sup>cf. Python Docs: `dict.get()` method

```
>>> greeting(333333)
'Hi there!'
```

Our final implementation of `greeting()` is concise, clean, and only uses features from the Python standard library. Therefore, I believe it is the best solution for this particular situation.

## Key Takeaways

- Avoid explicit *key in dict* checks when testing for membership.
- EAFP-style exception handling or using the built-in `get()` method is preferable.
- In some cases, the `collections.defaultdict` class from the standard library can also be helpful.

## 7.2 Sorting Dictionaries for Fun and Profit

Python dictionaries don't have an inherent order. You can iterate over them just fine but there's no guarantee that iteration returns the dictionary's elements in any particular order (although this is changing with Python 3.6).

However, it's frequently useful to get a *sorted representation* of a dictionary to put the dictionary's items into an arbitrary order based on their key, value, or some other derived property. Suppose you have a dictionary `xs` with the following key/value pairs:

```
>>> xs = {'a': 4, 'c': 2, 'b': 3, 'd': 1}
```

To get a sorted list of the key/value pairs in this dictionary, you could use the dictionary's `items()` method and then sort the resulting sequence in a second pass:

```
>>> sorted(xs.items())  
[('a', 4), ('b', 3), ('c', 2), ('d', 1)]
```

The key/value tuples are ordered using Python's standard lexicographical ordering for comparing sequences.

To compare two tuples, Python compares the items stored at index zero first. If they differ, this defines the outcome of the comparison. If they're equal, the next two items at index one are compared, and so on.

Now, because we took these tuples from a dictionary, all of the former dictionary keys at index zero in each tuple are unique. Therefore, there are no ties to break here.

In some cases a lexicographical ordering might be exactly what you want. In other cases you might want to sort a dictionary by value instead.

Luckily, there's a way you can get complete control over how items are ordered. You can control the ordering by passing a *key func* to `sorted()` that will change how dictionary items are compared.

A *key func* is simply a normal Python function to be called on each element prior to making comparisons. The key func gets a dictionary item as its input and returns the desired “key” for the sort order comparisons.

Unfortunately, the word “key” is used in two contexts simultaneously here—the key func doesn't deal with dictionary keys, it merely maps each input item to an arbitrary *comparison key*.

Now, maybe we should look at an example. Trust me, key funcs will be much easier to understand once you see some real code.

Let's say you wanted to get a sorted representation of a dictionary based on its *values*. To get this result you could use the following key func which returns the value of each key/value pair by looking up the second element in the tuple:

```
>>> sorted(xs.items(), key=lambda x: x[1])  
[('d', 1), ('c', 2), ('b', 3), ('a', 4)]
```

See how the resulting list of key/value pairs is now sorted by the values stored in the original dictionary? It's worth spending some time wrapping your head around how key funcs work. It's a powerful concept that you can apply in all kinds of Python contexts.

In fact, the concept is so common that Python's standard library includes the `operator` module. This module implements some of the most frequently used key funcs as plug-and-play building blocks, like `operator.itemgetter` and `operator.attrgetter`.

Here's an example of how you might replace the lambda-based index lookup in the first example with `operator.itemgetter`:

```
>>> import operator
>>> sorted(xs.items(), key=operator.itemgetter(1))
[('d', 1), ('c', 2), ('b', 3), ('a', 4)]
```

Using the `operator` module might communicate your code's intent more clearly in some cases. On the other hand, using a simple lambda expression might be just as readable and more explicit. In this particular case, I actually prefer the lambda expression.

Another benefit of using lambdas as a custom key func is that you get to control the sort order in much finer detail. For example, you could sort a dictionary based on the absolute numeric value of each value stored in it:

```
>>> sorted(xs.items(), key=lambda x: abs(x[1]))
```

If you need to reverse the sort order so that larger values go first, you can use the `reverse=True` keyword argument when calling `sorted()`:

```
>>> sorted(xs.items(),
           key=lambda x: x[1],
           reverse=True)
[('a', 4), ('b', 3), ('c', 2), ('d', 1)]
```

Like I said earlier, it's totally worth spending some time getting a good grip on how key funcs work in Python. They provide you with a ton of flexibility and can often save you from writing code to transform one data structure into another.

## Key Takeaways

- When creating sorted “views” of dictionaries and other collections, you can influence the sort order with a *key func*.

- *Key funcs* are an important concept in Python. The most frequently used ones were even added to the `operator` module in the standard library.
- Functions are first-class citizens in Python. This is a powerful feature you'll find used everywhere in the language.



## 7.3 Emulating Switch/Case Statements With Dicts

Python doesn't have switch/case statements so it's sometimes necessary to write long `if...elif...else` chains as a workaround. In this chapter you'll discover a trick you can use to emulate switch/case statements in Python with dictionaries and first-class functions. Sound exciting? Great—here we go!

Imagine we had the following `if`-chain in our program:

```
>>> if cond == 'cond_a':  
...     handle_a()  
... elif cond == 'cond_b':  
...     handle_b()  
... else:  
...     handle_default()
```

Of course, with only three different conditions, this isn't too horrible yet. But just imagine if we had ten or more `elif` branches in this statement. Things would start to look a little different. I consider long `if`-chains to be a *code smell* that makes programs more difficult to read and maintain.

One way to deal with long `if...elif...else` statements is to replace them with dictionary lookup tables that emulate the behavior of switch/case statements.

The idea here is to leverage the fact that Python has *first-class functions*. This means they can be passed as arguments to other functions, returned as values from other functions, and assigned to variables and stored in data structures.

For example, we can define a function and then store it in a list for later access:

```
>>> def myfunc(a, b):  
...     return a + b  
...  
>>> funcs = [myfunc]  
>>> funcs[0]  
<function myfunc at 0x107012230>
```

The syntax for calling this function works as you’d intuitively expect—we simply use an index into the list and then use the “()” call syntax for calling the function and passing arguments to it:

```
>>> funcs[0](2, 3)  
5
```

Now, how are we going to use first-class functions to cut our chained if-statement back to size? The core idea here is to define a dictionary that maps lookup keys for the input conditions to functions that will carry out the intended operations:

```
>>> func_dict = {  
...     'cond_a': handle_a,  
...     'cond_b': handle_b  
... }
```

Instead of filtering through the if-statement, checking each condition as we go along, we can do a dictionary key lookup to get the handler function and then call it:

```
>>> cond = 'cond_a'  
>>> func_dict[cond]()
```

This implementation already sort-of works, at least as long as `cond` can be found in the dictionary. If it’s not in there, we’ll get a `KeyError` exception.

So let's look for a way to support a *default* case that would match the original *else* branch. Luckily all Python dicts have a `get()` method on them that returns the value for a given key, or a default value if the key can't be found. This is exactly what we need here:

```
>>> func_dict.get(cond, handle_default)()
```

This code snippet might look syntactically odd at first, but when you break it down, it works exactly like the earlier example. Again, we're using Python's first-class functions to pass `handle_default` to the `get()`-lookup as a fallback value. That way, if the condition can't be found in the dictionary, we avoid raising a `KeyError` and call the default handler function instead.

Let's take a look at a more complete example for using dictionary lookups and first-class functions to replace *if*-chains. After reading through the following example, you'll be able to see the pattern needed to transform certain kinds of *if*-statements to a dictionary-based dispatch.

We're going to write another function with an *if*-chain that we'll then transform. The function takes a string opcode like "add" or "mul" and then does some math on the operands *x* and *y*:

```
>>> def dispatch_if(operator, x, y):  
...     if operator == 'add':  
...         return x + y  
...     elif operator == 'sub':  
...         return x - y  
...     elif operator == 'mul':  
...         return x * y  
...     elif operator == 'div':  
...         return x / y
```

To be honest, this is yet another toy example (I don't want to bore you with pages and pages of code here), but it'll serve well to illustrate the

underlying design pattern. Once you “get” the pattern, you’ll be able to apply it in all kinds of different scenarios.

You can try out this `dispatch_if()` function to perform simple calculations by calling the function with a string opcode and two numeric operands:

```
>>> dispatch_if('mul', 2, 8)
16
>>> dispatch_if('unknown', 2, 8)
None
```

Please note that the 'unknown' case works because Python adds an implicit `return None` statement to the end of any function.

So far so good. Let’s transform the original `dispatch_if()` into a new function which uses a dictionary to map opcodes to arithmetic operations with first-class functions.

```
>>> def dispatch_dict(operator, x, y):
...     return {
...         'add': lambda: x + y,
...         'sub': lambda: x - y,
...         'mul': lambda: x * y,
...         'div': lambda: x / y,
...     }.get(operator, lambda: None)()
```

This dictionary-based implementation gives the same results as the original `dispatch_if()`. We can call both functions in exactly the same way:

```
>>> dispatch_dict('mul', 2, 8)
16
>>> dispatch_dict('unknown', 2, 8)
None
```

There are a couple of ways this code could be further improved if it was real “production-grade” code.

First of all, every time we call `dispatch_dict()`, it creates a temporary dictionary and a bunch of lambdas for the opcode lookup. This isn’t ideal from a performance perspective. For code that needs to be fast, it makes more sense to create the dictionary once as a constant and then to reference it when the function is called. We don’t want to recreate the dictionary every time we need to do a lookup.

Second, if we really wanted to do some simple arithmetic like  $x + y$ , then we’d be better off using Python’s built-in operator module instead of the lambda functions used in the example. The operator module provides implementations for all of Python’s operators, for example `operator.mul`, `operator.div`, and so on. This is a minor point, though. I intentionally used lambdas in this example to make it more generic. This should help you apply the pattern in other situations as well.

Well, now you’ve got another tool in your bag of tricks that you can use to simplify some of your `if`-chains should they get unwieldy. Just remember—this technique won’t apply in every situation and sometimes you’ll be better off with a plain `if`-statement.

## Key Takeaways

- Python doesn’t have a switch/case statement. But in some cases you can avoid long `if`-chains with a dictionary-based dispatch table.
- Once again Python’s first-class functions prove to be a powerful tool. But with great power comes great responsibility.

## 7.4 The Craziest Dict Expression in the West

Sometimes you strike upon a tiny code example that has real depth to it—a single line of code that can teach you a lot about a programming language if you ponder it enough. Such a code snippet feels like a *Zen kōan*: a question or statement used in Zen practice to provoke doubt and test the student’s progress.

The tiny little code snippet we’ll discuss in this chapter is one such example. Upon first glance, it might seem like a straightforward dictionary expression, but when considered at close range, it takes you on a mind-expanding journey through the CPython interpreter.

I get such a kick out of this little one-liner that at one point I had it printed on my Python conference badges as a conversation starter. It also led to some rewarding conversations with members of my Python newsletter.

So without further ado, here is the code snippet. Take a moment to reflect on the following dictionary expression and what it will evaluate to:

```
>>> {True: 'yes', 1: 'no', 1.0: 'maybe'}
```

I’ll wait here...

Ok, ready?

This is the result we get when evaluating the above dict expression in a CPython interpreter session:

```
>>> {True: 'yes', 1: 'no', 1.0: 'maybe'}  
{True: 'maybe'}
```

I’ll admit I was pretty surprised about this result the first time I saw it. But it all makes sense when you investigate what happens, step

by step. So, let's think about why we get this—I want to say *slightly unintuitive*—result.

When Python processes our dictionary expression, it first constructs a new empty dictionary object; and then it assigns the keys and values to it in the order given in the dict expression.

Therefore, when we break it down, our dict expression is equivalent to this sequence of statements that are executed in order:

```
>>> xs = dict()
>>> xs[True] = 'yes'
>>> xs[1] = 'no'
>>> xs[1.0] = 'maybe'
```

Oddly enough, Python considers all dictionary keys used in this example to be *equal*:

```
>>> True == 1 == 1.0
True
```

Okay, but wait a minute here. I'm sure you can intuitively accept that `1.0 == 1`, but why would `True` be considered equal to 1 as well? The first time I saw this dictionary expression it really stumped me.

After doing some digging in the Python documentation, I learned that Python treats `bool` as a subclass of `int`. This is the case in Python 2 and Python 3:

“The Boolean type is a subtype of the integer type, and Boolean values behave like the values 0 and 1, respectively, in almost all contexts, the exception being that when converted to a string, the strings ‘False’ or ‘True’ are returned, respectively.”<sup>3</sup>

---

<sup>3</sup>cf. Python Docs: “The Standard Type Hierarchy”

And yes, this means you can *technically* use bools as indexes into a list or tuple in Python:

```
>>> ['no', 'yes'][True]
'yes'
```

But you probably should *not* use boolean variables like that for the sake of clarity (and the sanity of your colleagues.)

Anyway, let's come back to our dictionary expression.

As far as Python is concerned, True, 1, and 1.0 all represent *the same dictionary key*. As the interpreter evaluates the dictionary expression, it repeatedly overwrites the value for the key True. This explains why, in the end, the resulting dictionary only contains a single key.

Before we move on, let's have another look at the original dictionary expression:

```
>>> {True: 'yes', 1: 'no', 1.0: 'maybe'}
{True: 'maybe'}
```

Why do we still get True as the key here? Shouldn't the key also change to 1.0 at the end, due to the repeated assignments?

After some more research in the CPython interpreter source code, I learned that Python's dictionaries don't update the key object itself when a new value is associated with it:

```
>>> ys = {1.0: 'no'}
>>> ys[True] = 'yes'
>>> ys
{1.0: 'yes'}
```

Of course this makes sense as a performance optimization—if the keys are considered identical, then why spend time updating the original?



In the last example you saw that the initial `True` object is never replaced as the key. Therefore, the dictionary's string representation still prints the key as `True` (instead of `1` or `1.0`.)

With what we know now, it looks like the values in the resulting dict are getting overwritten only because they compare as equal. However, it turns out that this effect isn't caused by the `__eq__` equality check alone, either.

Python dictionaries are backed by a hash table data structure. When I first saw this surprising dictionary expression, my hunch was that this behavior had something to do with hash collisions.

You see, a hash table internally stores the keys it contains in different "buckets" according to each key's hash value. The hash value is derived from the key as a numeric value of a fixed length that uniquely identifies the key.

This allows for fast lookups. It's much quicker to search for a key's numeric hash value in a lookup table instead of comparing the full key object against all other keys and checking for equality.

However, the way hash values are typically calculated isn't perfect. And eventually, two or more keys that are actually different will have the same derived hash value, and they will end up in the same lookup table bucket.

If two keys have the same hash value, that's called a *hash collision*, and it's a special case that the hash table's algorithms for inserting and finding elements need to handle.

Based on that assessment, it's fairly likely that hashing has something to do with the surprising result we got from our dictionary expression. So let's find out if the keys' hash values also play a role here.

I'm defining the following class as our little detective tool:

```
class AlwaysEquals:
    def __eq__(self, other):
        return True

    def __hash__(self):
        return id(self)
```

This class is special in two ways.

First, because its `__eq__` dunder method always returns `True`, all instances of this class will pretend they're equal to *any* other object:

```
>>> AlwaysEquals() == AlwaysEquals()
True
>>> AlwaysEquals() == 42
True
>>> AlwaysEquals() == 'waaat?'
True
```

And second, each `AlwaysEquals` instance will also return a unique hash value generated by the built-in `id()` function:

```
>>> objects = [AlwaysEquals(),
                AlwaysEquals(),
                AlwaysEquals()]
>>> [hash(obj) for obj in objects]
[4574298968, 4574287912, 4574287072]
```

In CPython, `id()` returns the address of the object in memory, which is guaranteed to be unique.

With this class we can now create objects that pretend to be equal to any other object but have a unique hash value associated with them. That'll allow us to test if dictionary keys are overwritten based on their equality comparison result alone.

And, as you can see, the keys in the next example are *not* getting overwritten, even though they always compare as equal:

```
>>> {AlwaysEquals(): 'yes', AlwaysEquals(): 'no'}
{ <AlwaysEquals object at 0x110a3c588>: 'yes',
  <AlwaysEquals object at 0x110a3cf98>: 'no' }
```

We can also flip this idea around and check to see if returning the same hash value is enough to cause keys to get overwritten:

```
class SameHash:
    def __hash__(self):
        return 1
```

Instances of this SameHash class will compare as non-equal with each other but they will all share the same hash value of 1:

```
>>> a = SameHash()
>>> b = SameHash()
>>> a == b
False
>>> hash(a), hash(b)
(1, 1)
```

Let's look at how Python's dictionaries react when we attempt to use instances of the SameHash class as dictionary keys:

```
>>> {a: 'a', b: 'b'}
{ <SameHash instance at 0x7f7159020cb0>: 'a',
  <SameHash instance at 0x7f7159020cf8>: 'b' }
```

As this example shows, the “keys get overwritten” effect isn't caused by hash value collisions alone either.

Dictionaries check for equality and compare the hash value to determine if two keys are the same. Let's try and summarize the findings of our investigation:

The `{True: 'yes', 1: 'no', 1.0: 'maybe'}` dictionary expression evaluates to `{True: 'maybe'}` because the keys `True`, `1`, and `1.0` all compare as equal, *and* they all have the same hash value:

```
>>> True == 1 == 1.0
True
>>> (hash(True), hash(1), hash(1.0))
(1, 1, 1)
```

Perhaps not-so-surprising anymore, that's how we ended up with this result as the dictionary's final state:

```
>>> {True: 'yes', 1: 'no', 1.0: 'maybe'}
{True: 'maybe'}
```

We touched on a lot of subjects here, and this particular Python Trick can be a bit mind-boggling at first—that's why I compared it to a Zen kōan in the beginning.

If it's difficult to understand what's going on in this chapter, try playing through the code examples one by one in a Python interpreter session. You'll be rewarded with an expanded knowledge of Python's internals.

## Key Takeaways

- Dictionaries treat keys as identical if their `__eq__` comparison result says they're equal and their hash values are the same.
- Unexpected dictionary key collisions can and will lead to surprising results.

## 7.5 So Many Ways to Merge Dictionaries

Have you ever built a configuration system for one of your Python programs? A common use case for such systems is to take a data structure with default configuration options, and then to allow the defaults to be overridden selectively from user input or some other config source.

I often found myself using dictionaries as the underlying data structure for representing configuration keys and values. And so I frequently needed a way to combine or to *merge* the config defaults and the user overrides into a single dictionary with the final configuration values.

Or, to generalize: sometimes you need a way to merge two or more dictionaries into one, so that the resulting dictionary contains a combination of the keys and values of the source dicts.

In this chapter I'll show you a couple of ways to achieve that. Let's look at a simple example first so we have something to discuss. Imagine you had these two source dictionaries:

```
>>> xs = {'a': 1, 'b': 2}
>>> ys = {'b': 3, 'c': 4}
```

Now, you want to create a new dict `zs` that contains all of the keys and values of `xs` and all of the keys and values of `ys`. Also, if you read the example closely, you saw that the string `'b'` appears as a key in both dicts—we'll need to think about a conflict resolution strategy for duplicate keys as well.

The classical solution for the “merging multiple dictionaries” problem in Python is to use the built-in dictionary `update()` method:

```
>>> zs = {}
>>> zs.update(xs)
>>> zs.update(ys)
```

If you're curious, a naive implementation of `update()` might look something like this. We simply iterate over all of the items of the right-hand side dictionary and add each key/value pair to the left-hand side dictionary, overwriting existing keys as we go along:

```
def update(dict1, dict2):
    for key, value in dict2.items():
        dict1[key] = value
```

This results in a new dictionary `zs` which now contains the keys defined in `xs` and `ys`:

```
>>> zs
>>> {'c': 4, 'a': 1, 'b': 3}
```

You'll also see that the order in which we call `update()` determines how conflicts are resolved. The last update wins and the duplicate key `'b'` is associated with the value 3 that came from `ys`, the second source dictionary.

Of course you could expand this chain of `update()` calls for as long as you like in order to merge any number of dictionaries into one. It's a practical and well-readable solution that works in Python 2 and Python 3.

Another technique that works in Python 2 and Python 3 uses the `dict()` built-in combined with the `**`-operator for “unpacking” objects:

```
>>> zs = dict(xs, **ys)
>>> zs
{'a': 1, 'c': 4, 'b': 3}
```

However, just like making repeated `update()` calls, this approach only works for merging *two* dictionaries and cannot be generalized to combine an arbitrary number of dictionaries in one step.

Starting with Python 3.5, the `**`-operator became more flexible.<sup>4</sup> So in Python 3.5+ there's another—and arguably prettier—way to merge an arbitrary number of dictionaries:

```
>>> zs = {**xs, **ys}
```

This expression has the exact same result as a chain of `update()` calls. Keys and values are set in a left-to-right order, so we get the same conflict resolution strategy: the right-hand side takes priority, and a value in `ys` overrides any existing value under the same key in `xs`. This becomes clear when we look at the dictionary that results from the merge operation:

```
>>> zs
>>> {'c': 4, 'a': 1, 'b': 3}
```

Personally, I like the terseness of this new syntax and how it still remains sufficiently readable. There's always a fine balance between verbosity and terseness to keep the code as readable and maintainable as possible.

In this case, I'm leaning towards using the new syntax if I'm working with Python 3. Using the `**`-operator is also faster than using chained `update()` calls, which is yet another benefit.

## Key Takeaways

- In Python 3.5 and above you can use the `**`-operator to merge multiple dictionary objects into one with a single expression, overwriting existing keys left-to-right.
- To stay compatible with older versions of Python, you might want to use the built-in dictionary `update()` method instead.

---

<sup>4</sup>cf. PEP 448: “Additional Unpacking Generalizations”

## 7.6 Dictionary Pretty-Printing

Have you ever tried hunting down a bug in one of your programs by sprinkling a bunch of debug “print” statements to trace the execution flow? Or maybe you needed to generate a log message to print some configuration settings...

I have—and I’ve often been frustrated with how difficult some data structures are to read in Python when they’re printed as text strings. For example, here’s a simple dictionary. Printed in an interpreter session, the key order is arbitrary, and there’s no indentation to the resulting string:

```
>>> mapping = {'a': 23, 'b': 42, 'c': 0xc0ffee}
>>> str(mapping)
{'b': 42, 'c': 12648430, 'a': 23}
```

Luckily there are some easy-to-use alternatives to a straight up *to-str* conversion that give a more readable result. One option is using Python’s built-in `json` module. You can use `json.dumps()` to pretty-print Python dicts with nicer formatting:

```
>>> import json
>>> json.dumps(mapping, indent=4, sort_keys=True)
```

```
{
    "a": 23,
    "b": 42,
    "c": 12648430
}
```

These settings result in a nicely indented string representation that also normalizes the order of the dictionary keys for better legibility.

While this looks nice and readable, it isn’t a perfect solution. Printing dictionaries with the `json` module only works with dicts that con-



tain primitive types—you'll run into trouble trying to print a dictionary that contains a non-primitive data type, like a function:

```
>>> json.dumps({all: 'yup'})  
TypeError: "keys must be a string"
```

Another downside of using `json.dumps()` is that it can't stringify complex data types, like sets:

```
>>> mapping['d'] = {1, 2, 3}  
>>> json.dumps(mapping)  
TypeError: "set([1, 2, 3]) is not JSON serializable"
```

Also, you might run into trouble with how Unicode text is represented—in some cases you won't be able to take the output from `json.dumps` and copy and paste it into a Python interpreter session to reconstruct the original dictionary object.

The classical solution to pretty-printing objects in Python is the built-in `pprint` module. Here's an example:

```
>>> import pprint  
>>> pprint.pprint(mapping)  
{'a': 23, 'b': 42, 'c': 12648430, 'd': set([1, 2, 3])}
```

You can see that `pprint` is able to print data types like sets, and it also prints the dictionary keys in a reproducible order. Compared to the standard string representation for dictionaries, what we get here is much easier on the eyes.

However, compared to `json.dumps()`, it doesn't represent nested structures as well visually. Depending on the circumstances, this can be an advantage or a disadvantage. I occasionally use `json.dumps()` to print dictionaries because of the improved readability and formatting, but only if I'm sure they're free of non-primitive data types.

### Key Takeaways

- The default to-string conversion for dictionary objects in Python can be difficult to read.
- The `pprint` and `json` module are “higher-fidelity” options built into the Python standard library.
- Be careful with using `json.dumps()` and non-primitive keys and values as this will trigger a `TypeError`.

# **Chapter 8**

## **Pythonic Productivity Techniques**

## 8.1 Exploring Python Modules and Objects

You can interactively explore modules and objects directly from the Python interpreter. This is an underrated feature that's easy to overlook, especially if you're switching to Python from another language.

Many programming languages make it difficult to inspect a package or class without consulting online documentation or learning interface definitions by heart.

Python is different—an effective developer will spend quite a bit of time in REPL sessions working interactively with the Python interpreter. For example, I often do this to work out little snippets of code and logic that I then copy and paste into a Python file I'm working on in my editor.

In this chapter you'll learn two simple techniques you can use to explore Python classes and methods interactively from the interpreter.

These techniques will work on any Python install—just start up the Python interpreter with the `python` command from the command-line and fire away. This is great for debugging sessions on systems where you don't have access to a fancy editor or IDE, for example, because you're working over the network in a terminal session.

Ready? Here we go! Imagine you're writing a program that uses Python's `datetime` module from the standard library. How can you find out what functions or classes this module exports, and which methods and attributes can you find on its classes?

Consulting a search engine or looking up the official Python documentation on the web would be one way to do it. But Python's built-in `dir()` function lets you access this information directly from the Python REPL:

```
>>> import datetime
>>> dir(datetime)
['MAXYEAR', 'MINYEAR', '__builtins__', '__cached__',
 '__doc__', '__file__', '__loader__', '__name__',
 '__package__', '__spec__', '_divide_and_round',
 'date', 'datetime', 'datetime_CAPI', 'time',
 'timedelta', 'timezone', 'tzinfo']
```

In the example above, I first imported the `datetime` module from the standard library and then inspected it with the `dir()` function. Calling `dir()` on a module gives you an alphabetized list of the names and attributes the module provides.

Because “everything” is an object in Python, the same technique works not only on the module itself, but also on the classes and data structures exported by the module.

In fact, you can keep drilling down into the module by calling `dir()` again on individual objects that look interesting. For example, here’s how you’d inspect the `datetime.date` class:

```
>>> dir(datetime.date)
['__add__', '__class__', ..., 'day', 'fromordinal',
 'isocalendar', 'isoformat', 'isoweekday', 'max',
 'min', 'month', 'replace', 'resolution', 'strftime',
 'timetuple', 'today', 'toordinal', 'weekday', 'year']
```

As you can see, `dir()` gives you a quick overview of what’s available on a module or class. If you don’t remember the exact spelling for a particular class or function, then maybe that’s all you need to keep going without interrupting your coding flow.

Sometimes calling `dir()` on an object will result in too much information—on a complex module or class you’ll get a long printout that’s difficult to read quickly. Here’s a little trick you can use to filter down the list of attributes to just the ones you’re interested in:

```
>>> [_ for _ in dir(datetime) if 'date' in _.lower()]
['date', 'datetime', 'datetime_CAPI']
```

Here, I used a list comprehension to filter down the results of the `dir(datetime)` call to only contain names that include the word “date.” Notice how I called the `lower()` method on each name to make sure the filtering is case-insensitive.

Getting a raw listing of the attributes on an object won’t always be enough information to solve the problem at hand. So, how can you get more info and further details on the functions and classes exported by the `datetime` module?

Python’s built-in `help()` function will come to your rescue. With it, you can invoke Python’s interactive help system to browse the auto-generated documentation for any Python object:

```
>>> help(datetime)
```

If you run the above example in a Python interpreter session, your terminal will display a text-based help screen for the `datetime` module, looking somewhat like this:

```
Help on module datetime:
```

```
NAME
```

```
    datetime - Fast implementation of the datetime type.
```

```
CLASSES
```

```
    builtins.object
```

```
        date
```

```
            datetime
```

```
        time
```

You can use the cursor up and down keys to scroll through the documentation. Alternatively you can also hit the space bar to scroll down

a few lines at once. To leave this interactive help mode you'll need to press the `q` key. This will take you back to the interpreter prompt. Nice feature, right?

By the way, you can call `help()` on arbitrary Python objects, including other built-in functions and your own Python classes. The Python interpreter automatically generates this documentation from the attributes on an object and its docstring (if available.) The examples below are all valid uses of the help function:

```
>>> help(datetime.date)
>>> help(datetime.date.fromtimestamp)
>>> help(dir)
```

Of course, `dir()` and `help()` won't replace nicely formatted HTML documentation, the powers of a search engine, or a Stack Overflow search. But they're great tools for quickly looking things up without having to switch away from the Python interpreter. They're also available offline and work without an internet connection—which can be very useful in a pinch.

### Key Takeaways

- Use the built-in `dir()` function to interactively explore Python modules and classes from an interpreter session.
- The `help()` built-in lets you browse through the documentation right from your interpreter (hit `q` to exit.)

## 8.2 Isolating Project Dependencies With Virtualenv

Python includes a powerful packaging system to manage the module dependencies of your programs. You’ve probably used it to install third-party packages with the `pip` package manager command.

One confusing aspect of installing packages with `pip` is that it tries to install them into your *global* Python environment by default.

Sure, this makes any new packages you install available globally on your system, which is great for convenience. But it also quickly turns into a nightmare if you’re working with multiple projects that require different versions of the *same* package.

For example, what if one of your projects needs version 1.3 of a library while another project needs version 1.4 of the same library?

When you install packages globally *there can be only one* version of a Python library across all of your programs. This means you’ll quickly run into version conflicts—just like the Highlander did.

And it gets worse. You might also have different programs that need different versions of Python itself. For example, some programs might still run on Python 2 while most of your new development happens in Python 3. Or, what if one of your projects needs Python 3.3, while everything else runs on Python 3.6?

Besides that, installing Python packages globally can also incur a security risk. Modifying the global environment often requires you to run the `pip install` command with superuser (root/admin) credentials. Because `pip` downloads and executes code from the internet when you install a new package, this is generally not recommended. Hopefully the code is trustworthy, but who knows what it will really do?



## Virtual Environments to the Rescue

The solution to these problems is to separate your Python environments with so-called *virtual environments*. They allow you to separate Python dependencies by project and give you the ability to select between different versions of the Python interpreter.

A *virtual environment* is an isolated Python environment. Physically, it lives inside a folder containing all the packages and other dependencies, like native-code libraries and the interpreter runtime, that a Python project needs. (Behind the scenes, those files might not be real copies but symbolic links to save memory.)

To demonstrate how virtual environments work, I'll give you a quick walkthrough where we'll set up a new environment (or *virtualenv*, as they're called for short) and then install a third-party package into it.

Let's first check where the global Python environment currently resides. On Linux or macOS, we can use the `which` command-line tool to look up the path to the `pip` package manager:

```
$ which pip3
/usr/local/bin/pip3
```

I usually put my virtual environments right into my project folders to keep them nice and separate. But you could also have a dedicated “python-environments” directory somewhere to hold all of your environments across projects. The choice is yours.

Let's create a new Python virtual environment:

```
$ python3 -m venv ./venv
```

This will take a moment and will create a new `venv` folder in the current directory and seed it with a baseline Python 3 environment:

```
$ ls venv/  
bin          include     lib         pyvenv.cfg
```

If you check the active version of `pip` (with the `which` command), you'll see it's still pointing to the global environment, `/usr/local/bin/pip3` in my case:

```
(venv) $ which pip3  
/usr/local/bin/pip3
```

This means if you install packages now, they'd still end up in the global Python environment. Creating a virtual environment folder isn't enough—you'll need to explicitly *activate* the new virtual environment so that future runs of the `pip` command reference it:

```
$ source ./venv/bin/activate  
(venv) $
```

Running the `activate` command configures your current shell session to use the Python and `pip` commands from the virtual environment instead.<sup>1</sup>

Notice how this changed your shell prompt to include the name of the active virtual environment inside parentheses: `(venv)`. Let's check which `pip` executable is active now:

```
(venv) $ which pip3  
/Users/dan/my-project/venv/bin/pip3
```

As you can see, running the `pip3` command would now run the version from the virtual environment and not the global one. The same

---

<sup>1</sup>On Windows there's an `activate` command you need to run directly instead of loading it with `source`.

is true for the Python interpreter executable. Running `python` from the command-line would now also load the interpreter from the `venv` folder:

```
(venv) $ which python
/Users/dan/my-project/venv/bin/python
```

Note that this is still a blank slate, a completely clean Python environment. Running `pip list` will show an almost empty list of installed packages that only includes the baseline modules necessary to support `pip` itself:

```
(venv) $ pip list
pip (9.0.1)
setuptools (28.8.0)
```

Let's go ahead and install a Python package into the virtual environment now. You'll want to use the familiar `pip install` command for that:

```
(venv) $ pip install schedule
Collecting schedule
  Downloading schedule-0.4.2-py2.py3-none-any.whl
Installing collected packages: schedule
Successfully installed schedule-0.4.2
```

You'll notice two important changes here. First, you won't need admin permissions to run this command any longer. And second, installing or updating a package with an active virtual environment means that all files will end up in a subfolder in the virtual environment's directory.

Therefore, your project dependencies will be physically separated from all other Python environments on your system, including the

global one. In effect, you get a clone of the Python runtime that’s dedicated to one project only.

By running `pip list` again, you can see that the `schedule` library was installed successfully into the new environment:

```
(venv) $ pip list
pip (9.0.1)
schedule (0.4.2)
setuptools (28.8.0)
```

If we spin up a Python interpreter session with the `python` command, or run a standalone `.py` file with it, it will use the Python interpreter and the dependencies installed into the virtual environment—as long as the environment is still active in the current shell session.

But how do you deactivate or “leave” a virtual environment again? Similar to the `activate` command, there’s a `deactivate` command that takes you back to the global environment:

```
(venv) $ deactivate
$ which pip3
/usr/local/bin
```

Using virtual environments will help keep your system uncluttered and your Python dependencies neatly organized. As a best practice, all of your Python projects should use virtual environments to keep their dependencies separate and to avoid version conflicts.

Understanding and using virtual environments also puts you on the right track to use more advanced dependency management methods like specifying project dependencies with `requirements.txt` files.

If you’re looking for a deep dive on this subject with additional productivity tips, be sure to check out my [Managing Python Dependencies course available on dbader.org](#).

### Key Takeaways

- Virtual environments keep your project dependencies separated. They help you avoid version conflicts between packages and different versions of the Python runtime.
- As a best practice, all of your Python projects should use virtual environments to store their dependencies. This will help avoid headaches.

## 8.3 Peeking Behind the Bytecode Curtain

When the CPython interpreter executes your program, it first translates it into a sequence of bytecode instructions. Bytecode is an intermediate language for the Python virtual machine that's used as a performance optimization.

Instead of directly executing the human-readable source code, compact numeric codes, constants, and references are used that represent the result of compiler parsing and semantic analysis.

This saves time and memory for repeated executions of programs or parts of programs. For example, the bytecode resulting from this compilation step is cached on disk in `.pyc` and `.pyo` files so that executing the same Python file is faster the second time around.

All of this is completely transparent to the programmer. You don't have to be aware that this intermediate translation step happens, or how the Python virtual machine deals with the bytecode. In fact, the bytecode format is deemed an implementation detail and not guaranteed to remain stable or compatible between Python versions.

And yet, I find it very enlightening to see *how the sausage is made* and to peek behind the abstractions provided by the CPython interpreter. Understanding at least some of the inner workings can help you write more performant code (when that's important). And it's also a lot of fun.

Let's take this simple `greet()` function as a lab sample we can play with and use to understand Python's bytecode:

```
def greet(name):  
    return 'Hello, ' + name + '!'
```

```
>>> greet('Guido')  
'Hello, Guido!'
```

Remember how I said that CPython first translates our source code into an intermediate language before it “runs” it? Well, if that’s true, we should be able to see the results of this compilation step. And we can.

Each function has a `__code__` attribute (in Python 3) that we can use to get at the virtual machine instructions, constants, and variables used by our `greet` function:

```
>>> greet.__code__.co_code
b'd\x01|\x00\x17\x00d\x02\x17\x00S\x00'
>>> greet.__code__.co_consts
(None, 'Hello, ', '!')
>>> greet.__code__.co_varnames
('name',)
```

You can see `co_consts` contains parts of the greeting string our function assembles. Constants and code are kept separate to save memory space. Constants are, well, constant—meaning they can never be modified and are used interchangeably in multiple places.

So instead of repeating the actual constant values in the `co_code` instruction stream, Python stores constants separately in a lookup table. The instruction stream can then refer to a constant with an index into the lookup table. The same is true for variables stored in the `co_varnames` field.

I hope this general concept is starting to become more clear. But looking at the `co_code` instruction stream still makes me feel a little queasy. This intermediate language is clearly meant to be easy to work with for the Python virtual machine, not humans. After all, that’s what the text-based source code is for.

The developers working on CPython realized that too. So they gave us another tool called a *disassembler* to make inspecting the bytecode easier.

Python’s bytecode disassembler lives in the `dis` module that’s part of the standard library. So we can just import it and call `dis.dis()` on our `greet` function to get a slightly easier-to-read representation of its bytecode:

```
>>> import dis
>>> dis.dis(greet)
 2          0 LOAD_CONST          1 ('Hello, ')
          2 LOAD_FAST            0 (name)
          4 BINARY_ADD
          6 LOAD_CONST          2 ('!')
          8 BINARY_ADD
         10 RETURN_VALUE
```

The main thing disassembling `dis` did was split up the instruction stream and give each *opcode* in it a human-readable name like `LOAD_CONST`.

You can also see how constant and variable references are now interleaved with the bytecode and printed in full to spare us the mental gymnastics of a `co_const` or `co_varnames` table lookup. Neat!

Looking at the human-readable opcodes, we can begin to understand how CPython represents and executes the `'Hello, ' + name + '!'` expression in the original `greet()` function.

It first retrieves the constant at index 1 (`'Hello, '`) and puts it on the *stack*. It then loads the contents of the `name` variable and also puts them on the *stack*.

The *stack* is the data structure used as internal working storage for the virtual machine. There are different classes of virtual machines and one of them is called a *stack machine*. CPython’s virtual machine is an implementation of such a stack machine. If the whole thing is named after the stack, you can imagine what a central role this data structure plays.

By the way—I’m only touching the surface here. If you’re interested in



this topic you'll find a book recommendation at the end of this chapter. Reading up on virtual machine theory is enlightening (and a ton of fun).

What's interesting about a *stack* as an abstract data structure is that, at the bare minimum, it only supports two operations: *push* and *pop*. *Push* adds a value to the top of the stack and *pop* removes and returns the topmost value. Unlike an array, there's no way to access elements "below" the top level.

I find it fascinating that such a simple data structure has so many uses. But I'm getting carried away again...

Let's assume the stack starts out empty. After the first two opcodes have been executed, this is what the contents of the VM stack look like (0 is the topmost element):

```
0: 'Guido' (contents of "name")
1: 'Hello, '
```

The `BINARY_ADD` instruction pops the two string values off the stack, concatenates them, and then pushes the result on the stack again:

```
0: 'Hello, Guido'
```

Then there's another `LOAD_CONST` to get the exclamation mark string on the stack:

```
0: '!'
1: 'Hello, Guido'
```

The next `BINARY_ADD` opcode again combines the two to generate the final greeting string:

```
0: 'Hello, Guido!'
```

The last bytecode instruction is `RETURN_VALUE` which tells the virtual machine that what's currently on top of the stack is the return value for this function so it can be passed on to the caller.

And voila, we just traced back how our `greet()` function gets executed internally by the CPython virtual machine. Isn't that cool?

There's much more to say about virtual machines, and this isn't the book for it. But if this got you interested, I highly recommend that you do some more reading on this fascinating subject.

It can be a lot of fun to define your own bytecode languages and to build little virtual machine experiments for them. A book on this topic that I'd recommend is *Compiler Design: Virtual Machines* by Wilhelm and Seidl.

## Key Takeaways

- CPython executes programs by first translating them into an intermediate bytecode and then running the bytecode on a stack-based virtual machine.
- You can use the built-in `dis` module to peek behind the scenes and inspect the bytecode.
- Study up on virtual machines—it's worth it.

# Chapter 9

## Closing Thoughts

Congratulations—you made it all the way to the end! Time to give yourself a pat on the back, since most people buy a book and never even crack it open or make it past the first chapter.

But now that you’ve read the book, this is where the real work starts—there’s a big difference between *reading* and *doing*. Take the new skills and tricks you learned in this book, and go out there and use them. Don’t let this be just another programming book you read.

What if you started sprinkling some of Python’s advanced features in your code from now on? A nice and clean generator expression here, an elegant use of the `with`-statement there...

You’ll catch the attention of your peers in no time—and in a good way too, if you do it right. With some practice you’ll have no trouble applying these advanced Python features tastefully, and to only use them where they make sense and help make your code more expressive.

And trust me, your colleagues will pick up on this after a while. If they ask you questions, be generous and helpful. Pull everyone around you *up* and help them learn what you know. Maybe you can even give a little presentation on “writing clean Python” for your coworkers a few weeks down the road. Feel free to use my examples from the book.

---

There's a difference between *doing* a great job as a Python developer, and *to be seen doing* a great job. Don't be afraid to stick your head out. If you share your skills and newfound knowledge with the people around you, your career will benefit greatly.

I follow the same mindset in my own career and projects. And so, I'm always looking for ways to improve this book and my other Python training materials. If you'd like to let me know about an error, or if you just have a question or want to offer some constructive feedback, then please email me at [mail@dbader.org](mailto:mail@dbader.org).

Happy Pythoning!

— Dan Bader

P.S. Come visit me on the web and continue your Python journey at [dbader.org](http://dbader.org) and on my [YouTube channel](#).

## 9.1 Free Weekly Tips for Python Developers

Are you looking for a weekly dose of Python development tips to improve your productivity and streamline your workflows? Good news—I'm running a free email newsletter for Python developers just like you.

The newsletter emails I send out are not your typical “here's a list of popular articles” flavor. Instead I aim for sharing at least one original thought per week in a (short) essay-style format.

If you'd like to see what all the fuss is about, then head on over to [dbader.org/newsletter](https://dbader.org/newsletter) and enter your email address in the signup form. I'm looking forward to meeting you!

## 9.2 PythonistaCafe: A Community for Python Developers

Mastering Python is *not* just about getting the books and courses to study. To be successful you also need a way to stay motivated and to grow your abilities in the long run.

Many Pythonistas I know are struggling with this. It's simply a lot less fun to build your Python skills completely alone.

If you're a self-taught developer with a non-technical day job, it's hard to grow your skills all by yourself. And with no coders in your personal peer group, there's nobody to encourage or support you in your endeavor of becoming a better developer.

Maybe you're already working as a developer, but no one else at your company shares your love for Python. It's frustrating when you can't share your learning progress with anyone or ask for advice when you feel stuck.

From personal experience, I know that existing online communities and social media don't do a great job at providing that support network either. Here are a few of the best, but they still leave a lot to be desired:

- *Stack Overflow* is for asking focused, one-off questions. It's hard to make a human connection with fellow commenters on the platform. Everything is about the facts, not the people. For example, moderators will freely edit other people's questions, answers, and comments. It feels more like a wiki than a forum.
- *Twitter* is like a virtual water cooler and great for "hanging out" but it's limited to 140 characters at a time—not great for discussing anything substantial. Also, if you're not constantly online, you'll miss out on most of the conversations. And if you *are* constantly online, your productivity takes a hit from the

never-ending stream of interruptions and notifications. Slack chat groups suffer from the same flaws.

- *Hacker News* is for discussing and commenting on tech news. It doesn't foster long-term relationships between commenters. It's also one of the most aggressive communities in tech right now with little moderation and a borderline toxic culture.
- *Reddit* takes a broader stance and encourages more “human” discussions than Stack Overflow's one-off Q&A format. But it's a huge public forum with millions of users and has all of the associated problems: toxic behavior, overbearing negativity, people lashing out at each other, jealousy, ... In short, all the “best” parts of the human behavior spectrum.

Eventually I realized that what holds so many developers back is their limited access to the global Python coding community. That's why I founded [PythonistaCafe](#), a peer-to-peer learning community for Python developers.



A good way to think of PythonistaCafe is to see it as a club of mutual improvement for Python enthusiasts:

Inside PythonistaCafe you'll interact with professional developers and hobbyists from all over the world who will share their experiences in a safe setting—so you can learn from them and avoid the same mistakes they've made.

Ask anything you want and it will remain private. You must have an active membership to read and write comments and as a paid community, trolling and offensive behavior are virtually nonexistent.

The people you meet on the inside are actively committed to improving their Python skills because membership in PythonistaCafe is invite-only. All prospective members are required to submit an application to make sure they're a good fit for the community.

You'll be involved in a community that understands you, and the skills and career you're building, and what you're trying to achieve. If you're trying to grow your Python skills but haven't found the support system you need, we're right there for you.

PythonistaCafe is built on a private forum platform where you can ask questions, get answers, and share your progress. We have members located all over the world and with a wide range of proficiency levels.

You can learn more about PythonistaCafe, our community values, and what we're all about at [\*\*www.pythonistacafe.com\*\*](http://www.pythonistacafe.com).



*Recipes for Mastering Python 3*

**3rd Edition**



# Python Cookbook™

**O'REILLY®**

*David Beazley & Brian K. Jones*

# Python Cookbook

If you need help writing programs in Python 3, or want to update older Python 2 code, this book is just the ticket. Packed with practical recipes written and tested with Python 3.3, this unique cookbook is for experienced Python programmers who want to focus on modern tools and idioms.

Inside, you'll find complete recipes for more than a dozen topics, covering the core Python language as well as tasks common to a wide variety of application domains. Each recipe contains code samples you can use in your projects right away, along with a discussion about how and why the solution works.

## Topics include:

- Data Structures and Algorithms
- Strings and Text
- Numbers, Dates, and Times
- Iterators and Generators
- Files and I/O
- Data Encoding and Processing
- Functions
- Classes and Objects
- Metaprogramming
- Modules and Packages
- Network and Web Programming
- Concurrency
- Utility Scripting and System Administration
- Testing, Debugging, and Exceptions
- C Extensions

**David Beazley**, an independent software developer, teaches programming courses for developers, scientists, and engineers. He's the author of the *Python Essential Reference* (Addison-Wesley), and has created several open-source Python packages.

**Brian K. Jones** is a system administrator in the Department of Computer Science at Princeton University.

US \$49.99

CAN \$52.99

ISBN: 978-1-449-34037-7



Twitter: @oreillymedia  
facebook.com/oreilly

**O'REILLY®**  
oreilly.com

THIRD EDITION

---

# Python Cookbook

*David Beazley and Brian K. Jones*

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

## **Python Cookbook, Third Edition**

by David Beazley and Brian K. Jones

Copyright © 2013 David Beazley and Brian Jones. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://my.safaribooksonline.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or [corporate@oreilly.com](mailto:corporate@oreilly.com).

**Editors:** Meghan Blanchette and Rachel Roumeliotis

**Indexer:** WordCo Indexing Services

**Production Editor:** Kristen Borg

**Cover Designer:** Karen Montgomery

**Copyeditor:** Jasmine Kwityn

**Interior Designer:** David Futato

**Proofreader:** BIM Proofreading Services

**Illustrator:** Robert Romano

May 2013: Third Edition

### **Revision History for the Third Edition:**

2013-05-08: First release

See <http://oreilly.com/catalog/errata.csp?isbn=9781449340377> for release details.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *Python Cookbook*, the image of a springhaas, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-449-34037-7

[LSI]

---

# Table of Contents

<b>Preface.....</b>	<b>xi</b>
<b>1. Data Structures and Algorithms.....</b>	<b>1</b>
1.1. Unpacking a Sequence into Separate Variables	1
1.2. Unpacking Elements from Iterables of Arbitrary Length	3
1.3. Keeping the Last N Items	5
1.4. Finding the Largest or Smallest N Items	7
1.5. Implementing a Priority Queue	8
1.6. Mapping Keys to Multiple Values in a Dictionary	11
1.7. Keeping Dictionaries in Order	12
1.8. Calculating with Dictionaries	13
1.9. Finding Commonalities in Two Dictionaries	15
1.10. Removing Duplicates from a Sequence while Maintaining Order	17
1.11. Naming a Slice	18
1.12. Determining the Most Frequently Occurring Items in a Sequence	20
1.13. Sorting a List of Dictionaries by a Common Key	21
1.14. Sorting Objects Without Native Comparison Support	23
1.15. Grouping Records Together Based on a Field	24
1.16. Filtering Sequence Elements	26
1.17. Extracting a Subset of a Dictionary	28
1.18. Mapping Names to Sequence Elements	29
1.19. Transforming and Reducing Data at the Same Time	32
1.20. Combining Multiple Mappings into a Single Mapping	33
<b>2. Strings and Text.....</b>	<b>37</b>
2.1. Splitting Strings on Any of Multiple Delimiters	37
2.2. Matching Text at the Start or End of a String	38
2.3. Matching Strings Using Shell Wildcard Patterns	40
2.4. Matching and Searching for Text Patterns	42

2.5. Searching and Replacing Text	45
2.6. Searching and Replacing Case-Insensitive Text	46
2.7. Specifying a Regular Expression for the Shortest Match	47
2.8. Writing a Regular Expression for Multiline Patterns	48
2.9. Normalizing Unicode Text to a Standard Representation	50
2.10. Working with Unicode Characters in Regular Expressions	52
2.11. Stripping Unwanted Characters from Strings	53
2.12. Sanitizing and Cleaning Up Text	54
2.13. Aligning Text Strings	57
2.14. Combining and Concatenating Strings	58
2.15. Interpolating Variables in Strings	61
2.16. Reformatting Text to a Fixed Number of Columns	64
2.17. Handling HTML and XML Entities in Text	65
2.18. Tokenizing Text	66
2.19. Writing a Simple Recursive Descent Parser	69
2.20. Performing Text Operations on Byte Strings	78
<b>3. Numbers, Dates, and Times.....</b>	<b>83</b>
3.1. Rounding Numerical Values	83
3.2. Performing Accurate Decimal Calculations	84
3.3. Formatting Numbers for Output	87
3.4. Working with Binary, Octal, and Hexadecimal Integers	89
3.5. Packing and Unpacking Large Integers from Bytes	90
3.6. Performing Complex-Valued Math	92
3.7. Working with Infinity and NaNs	94
3.8. Calculating with Fractions	96
3.9. Calculating with Large Numerical Arrays	97
3.10. Performing Matrix and Linear Algebra Calculations	100
3.11. Picking Things at Random	102
3.12. Converting Days to Seconds, and Other Basic Time Conversions	104
3.13. Determining Last Friday's Date	106
3.14. Finding the Date Range for the Current Month	107
3.15. Converting Strings into Datetimes	109
3.16. Manipulating Dates Involving Time Zones	110
<b>4. Iterators and Generators.....</b>	<b>113</b>
4.1. Manually Consuming an Iterator	113
4.2. Delegating Iteration	114
4.3. Creating New Iteration Patterns with Generators	115
4.4. Implementing the Iterator Protocol	117
4.5. Iterating in Reverse	119
4.6. Defining Generator Functions with Extra State	120

4.7. Taking a Slice of an Iterator	122
4.8. Skipping the First Part of an Iterable	123
4.9. Iterating Over All Possible Combinations or Permutations	125
4.10. Iterating Over the Index-Value Pairs of a Sequence	127
4.11. Iterating Over Multiple Sequences Simultaneously	129
4.12. Iterating on Items in Separate Containers	131
4.13. Creating Data Processing Pipelines	132
4.14. Flattening a Nested Sequence	135
4.15. Iterating in Sorted Order Over Merged Sorted Iterables	136
4.16. Replacing Infinite while Loops with an Iterator	138
<b>5. Files and I/O.....</b>	<b>141</b>
5.1. Reading and Writing Text Data	141
5.2. Printing to a File	144
5.3. Printing with a Different Separator or Line Ending	144
5.4. Reading and Writing Binary Data	145
5.5. Writing to a File That Doesn't Already Exist	147
5.6. Performing I/O Operations on a String	148
5.7. Reading and Writing Compressed Datafiles	149
5.8. Iterating Over Fixed-Sized Records	151
5.9. Reading Binary Data into a Mutable Buffer	152
5.10. Memory Mapping Binary Files	153
5.11. Manipulating Pathnames	156
5.12. Testing for the Existence of a File	157
5.13. Getting a Directory Listing	158
5.14. Bypassing Filename Encoding	160
5.15. Printing Bad Filenames	161
5.16. Adding or Changing the Encoding of an Already Open File	163
5.17. Writing Bytes to a Text File	165
5.18. Wrapping an Existing File Descriptor As a File Object	166
5.19. Making Temporary Files and Directories	167
5.20. Communicating with Serial Ports	170
5.21. Serializing Python Objects	171
<b>6. Data Encoding and Processing.....</b>	<b>175</b>
6.1. Reading and Writing CSV Data	175
6.2. Reading and Writing JSON Data	179
6.3. Parsing Simple XML Data	183
6.4. Parsing Huge XML Files Incrementally	186
6.5. Turning a Dictionary into XML	189
6.6. Parsing, Modifying, and Rewriting XML	191
6.7. Parsing XML Documents with Namespaces	193

6.8. Interacting with a Relational Database	195
6.9. Decoding and Encoding Hexadecimal Digits	197
6.10. Decoding and Encoding Base64	199
6.11. Reading and Writing Binary Arrays of Structures	199
6.12. Reading Nested and Variable-Sized Binary Structures	203
6.13. Summarizing Data and Performing Statistics	214
<b>7. Functions.....</b>	<b>217</b>
7.1. Writing Functions That Accept Any Number of Arguments	217
7.2. Writing Functions That Only Accept Keyword Arguments	219
7.3. Attaching Informational Metadata to Function Arguments	220
7.4. Returning Multiple Values from a Function	221
7.5. Defining Functions with Default Arguments	222
7.6. Defining Anonymous or Inline Functions	224
7.7. Capturing Variables in Anonymous Functions	225
7.8. Making an N-Argument Callable Work As a Callable with Fewer Arguments	227
7.9. Replacing Single Method Classes with Functions	231
7.10. Carrying Extra State with Callback Functions	232
7.11. Inlining Callback Functions	235
7.12. Accessing Variables Defined Inside a Closure	238
<b>8. Classes and Objects.....</b>	<b>243</b>
8.1. Changing the String Representation of Instances	243
8.2. Customizing String Formatting	245
8.3. Making Objects Support the Context-Management Protocol	246
8.4. Saving Memory When Creating a Large Number of Instances	248
8.5. Encapsulating Names in a Class	250
8.6. Creating Managed Attributes	251
8.7. Calling a Method on a Parent Class	256
8.8. Extending a Property in a Subclass	260
8.9. Creating a New Kind of Class or Instance Attribute	264
8.10. Using Lazily Computed Properties	267
8.11. Simplifying the Initialization of Data Structures	270
8.12. Defining an Interface or Abstract Base Class	274
8.13. Implementing a Data Model or Type System	277
8.14. Implementing Custom Containers	283
8.15. Delegating Attribute Access	287
8.16. Defining More Than One Constructor in a Class	291
8.17. Creating an Instance Without Invoking <i>init</i>	293
8.18. Extending Classes with Mixins	294
8.19. Implementing Stateful Objects or State Machines	299



8.20. Calling a Method on an Object Given the Name As a String	305
8.21. Implementing the Visitor Pattern	306
8.22. Implementing the Visitor Pattern Without Recursion	311
8.23. Managing Memory in Cyclic Data Structures	317
8.24. Making Classes Support Comparison Operations	321
8.25. Creating Cached Instances	323
<b>9. Metaprogramming.....</b>	<b>329</b>
9.1. Putting a Wrapper Around a Function	329
9.2. Preserving Function Metadata When Writing Decorators	331
9.3. Unwrapping a Decorator	333
9.4. Defining a Decorator That Takes Arguments	334
9.5. Defining a Decorator with User Adjustable Attributes	336
9.6. Defining a Decorator That Takes an Optional Argument	339
9.7. Enforcing Type Checking on a Function Using a Decorator	341
9.8. Defining Decorators As Part of a Class	345
9.9. Defining Decorators As Classes	347
9.10. Applying Decorators to Class and Static Methods	350
9.11. Writing Decorators That Add Arguments to Wrapped Functions	352
9.12. Using Decorators to Patch Class Definitions	355
9.13. Using a Metaclass to Control Instance Creation	356
9.14. Capturing Class Attribute Definition Order	359
9.15. Defining a Metaclass That Takes Optional Arguments	362
9.16. Enforcing an Argument Signature on *args and **kwargs	364
9.17. Enforcing Coding Conventions in Classes	367
9.18. Defining Classes Programmatically	370
9.19. Initializing Class Members at Definition Time	374
9.20. Implementing Multiple Dispatch with Function Annotations	376
9.21. Avoiding Repetitive Property Methods	382
9.22. Defining Context Managers the Easy Way	384
9.23. Executing Code with Local Side Effects	386
9.24. Parsing and Analyzing Python Source	388
9.25. Disassembling Python Byte Code	392
<b>10. Modules and Packages.....</b>	<b>397</b>
10.1. Making a Hierarchical Package of Modules	397
10.2. Controlling the Import of Everything	398
10.3. Importing Package Submodules Using Relative Names	399
10.4. Splitting a Module into Multiple Files	401
10.5. Making Separate Directories of Code Import Under a Common Namespace	404
10.6. Reloading Modules	406

10.7. Making a Directory or Zip File Runnable As a Main Script	407
10.8. Reading Datafiles Within a Package	408
10.9. Adding Directories to sys.path	409
10.10. Importing Modules Using a Name Given in a String	411
10.11. Loading Modules from a Remote Machine Using Import Hooks	412
10.12. Patching Modules on Import	428
10.13. Installing Packages Just for Yourself	431
10.14. Creating a New Python Environment	432
10.15. Distributing Packages	433
<b>11. Network and Web Programming.....</b>	<b>437</b>
11.1. Interacting with HTTP Services As a Client	437
11.2. Creating a TCP Server	441
11.3. Creating a UDP Server	445
11.4. Generating a Range of IP Addresses from a CIDR Address	447
11.5. Creating a Simple REST-Based Interface	449
11.6. Implementing a Simple Remote Procedure Call with XML-RPC	454
11.7. Communicating Simply Between Interpreters	456
11.8. Implementing Remote Procedure Calls	458
11.9. Authenticating Clients Simply	461
11.10. Adding SSL to Network Services	464
11.11. Passing a Socket File Descriptor Between Processes	470
11.12. Understanding Event-Driven I/O	475
11.13. Sending and Receiving Large Arrays	481
<b>12. Concurrency.....</b>	<b>485</b>
12.1. Starting and Stopping Threads	485
12.2. Determining If a Thread Has Started	488
12.3. Communicating Between Threads	491
12.4. Locking Critical Sections	497
12.5. Locking with Deadlock Avoidance	500
12.6. Storing Thread-Specific State	504
12.7. Creating a Thread Pool	505
12.8. Performing Simple Parallel Programming	509
12.9. Dealing with the GIL (and How to Stop Worrying About It)	513
12.10. Defining an Actor Task	516
12.11. Implementing Publish/Subscribe Messaging	520
12.12. Using Generators As an Alternative to Threads	524
12.13. Polling Multiple Thread Queues	531
12.14. Launching a Daemon Process on Unix	534
<b>13. Utility Scripting and System Administration.....</b>	<b>539</b>

13.1. Accepting Script Input via Redirection, Pipes, or Input Files	539
13.2. Terminating a Program with an Error Message	540
13.3. Parsing Command-Line Options	541
13.4. Prompting for a Password at Runtime	544
13.5. Getting the Terminal Size	545
13.6. Executing an External Command and Getting Its Output	545
13.7. Copying or Moving Files and Directories	547
13.8. Creating and Unpacking Archives	549
13.9. Finding Files by Name	550
13.10. Reading Configuration Files	552
13.11. Adding Logging to Simple Scripts	555
13.12. Adding Logging to Libraries	558
13.13. Making a Stopwatch Timer	559
13.14. Putting Limits on Memory and CPU Usage	561
13.15. Launching a Web Browser	563
<b>14. Testing, Debugging, and Exceptions. ....</b>	<b>565</b>
14.1. Testing Output Sent to stdout	565
14.2. Patching Objects in Unit Tests	567
14.3. Testing for Exceptional Conditions in Unit Tests	570
14.4. Logging Test Output to a File	572
14.5. Skipping or Anticipating Test Failures	573
14.6. Handling Multiple Exceptions	574
14.7. Catching All Exceptions	576
14.8. Creating Custom Exceptions	578
14.9. Raising an Exception in Response to Another Exception	580
14.10. Reraising the Last Exception	582
14.11. Issuing Warning Messages	583
14.12. Debugging Basic Program Crashes	585
14.13. Profiling and Timing Your Program	587
14.14. Making Your Programs Run Faster	590
<b>15. C Extensions. ....</b>	<b>597</b>
15.1. Accessing C Code Using ctypes	599
15.2. Writing a Simple C Extension Module	605
15.3. Writing an Extension Function That Operates on Arrays	609
15.4. Managing Opaque Pointers in C Extension Modules	612
15.5. Defining and Exporting C APIs from Extension Modules	614
15.6. Calling Python from C	619
15.7. Releasing the GIL in C Extensions	625
15.8. Mixing Threads from C and Python	625
15.9. Wrapping C Code with Swig	627

15.10. Wrapping Existing C Code with Cython	632
15.11. Using Cython to Write High-Performance Array Operations	638
15.12. Turning a Function Pointer into a Callable	643
15.13. Passing NULL-Terminated Strings to C Libraries	644
15.14. Passing Unicode Strings to C Libraries	648
15.15. Converting C Strings to Python	653
15.16. Working with C Strings of Dubious Encoding	654
15.17. Passing Filenames to C Extensions	657
15.18. Passing Open Files to C Extensions	658
15.19. Reading File-Like Objects from C	659
15.20. Consuming an Iterable from C	662
15.21. Diagnosing Segmentation Faults	663
<b>A. Further Reading</b> .....	<b>665</b>
<b>Index</b> .....	<b>667</b>

---

# Preface

Since 2008, the Python world has been watching the slow evolution of Python 3. It was always known that the adoption of Python 3 would likely take a long time. In fact, even at the time of this writing (2013), most working Python programmers continue to use Python 2 in production. A lot has been made about the fact that Python 3 is not backward compatible with past versions. To be sure, backward compatibility is an issue for anyone with an existing code base. However, if you shift your view toward the future, you'll find that Python 3 offers much more than meets the eye.

Just as Python 3 is about the future, this edition of the *Python Cookbook* represents a major change over past editions. First and foremost, this is meant to be a very forward looking book. All of the recipes have been written and tested with Python 3.3 without regard to past Python versions or the “old way” of doing things. In fact, many of the recipes will only work with Python 3.3 and above. Doing so may be a calculated risk, but the ultimate goal is to write a book of recipes based on the most modern tools and idioms possible. It is hoped that the recipes can serve as a guide for people writing new code in Python 3 or those who hope to modernize existing code.

Needless to say, writing a book of recipes in this style presents a certain editorial challenge. An online search for Python recipes returns literally thousands of useful recipes on sites such as [ActiveState's Python recipes](#) or [Stack Overflow](#). However, most of these recipes are steeped in history and the past. Besides being written almost exclusively for Python 2, they often contain workarounds and hacks related to differences between old versions of Python (e.g., version 2.3 versus 2.4). Moreover, they often use outdated techniques that have simply become a built-in feature of Python 3.3. Finding recipes exclusively focused on Python 3 can be a bit more difficult.

Rather than attempting to seek out Python 3-specific recipes, the topics of this book are merely inspired by existing code and techniques. Using these ideas as a springboard, the writing is an original work that has been deliberately written with the most modern Python programming techniques possible. Thus, it can serve as a reference for anyone who wants to write their code in a modern style.

In choosing which recipes to include, there is a certain realization that it is simply impossible to write a book that covers every possible thing that someone might do with Python. Thus, a priority has been given to topics that focus on the core Python language as well as tasks that are common to a wide variety of application domains. In addition, many of the recipes aim to illustrate features that are new to Python 3 and more likely to be unknown to even experienced programmers using older versions. There is also a certain preference to recipes that illustrate a generally applicable programming technique (i.e., programming patterns) as opposed to those that narrowly try to address a very specific practical problem. Although certain third-party packages get coverage, a majority of the recipes focus on the core language and standard library.

## Who This Book Is For

This book is aimed at more experienced Python programmers who are looking to deepen their understanding of the language and modern programming idioms. Much of the material focuses on some of the more advanced techniques used by libraries, frameworks, and applications. Throughout the book, the recipes generally assume that the reader already has the necessary background to understand the topic at hand (e.g., general knowledge of computer science, data structures, complexity, systems programming, concurrency, C programming, etc.). Moreover, the recipes are often just skeletons that aim to provide essential information for getting started, but which require the reader to do more research to fill in the details. As such, it is assumed that the reader knows how to use search engines and Python's excellent online documentation.

Many of the more advanced recipes will reward the reader's patience with a much greater insight into how Python actually works under the covers. You will learn new tricks and techniques that can be applied to your own code.

## Who This Book Is Not For

This is not a book designed for beginners trying to learn Python for the first time. In fact, it already assumes that you know the basics that might be taught in a Python tutorial or more introductory book. This book is also not designed to serve as a quick reference manual (e.g., quickly looking up the functions in a specific module). Instead, the book aims to focus on specific programming topics, show possible solutions, and serve as a springboard for jumping into more advanced material you might find online or in a reference.

# Conventions Used in This Book

The following typographical conventions are used in this book:

## *Italic*

Indicates new terms, URLs, email addresses, filenames, and file extensions.

## Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

## Constant width bold

Shows commands or other text that should be typed literally by the user.

## *Constant width italic*

Shows text that should be replaced with user-supplied values or by values determined by context.



This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

# Online Code Examples

Almost all of the code examples in this book are available online at <http://github.com/dabeaz/python-cookbook>. The authors welcome bug fixes, improvements, and comments.

# Using Code Examples

This book is here to help you get your job done. In general, if this book includes code examples, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount

of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: *Python Cookbook*, 3rd edition, by David Beazley and Brian K. Jones (O'Reilly). Copyright 2013 David Beazley and Brian Jones, 978-1-449-34037-7.

If you feel your use of code examples falls outside fair use or the permission given here, feel free to contact us at [permissions@oreilly.com](mailto:permissions@oreilly.com).

## Safari® Books Online



*Safari Books Online* is an on-demand digital library that delivers expert **content** in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of **product mixes** and pricing programs for **organizations, government agencies, and individuals**. Subscribers have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and dozens **more**. For more information about Safari Books Online, please visit us **online**.

## How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.  
1005 Gravenstein Highway North  
Sebastopol, CA 95472  
800-998-9938 (in the United States or Canada)  
707-829-0515 (international or local)  
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at [http://oreil.ly/python\\_cookbook\\_3e](http://oreil.ly/python_cookbook_3e).



To comment or ask technical questions about this book, send email to [\*bookquestions@oreilly.com\*](mailto:bookquestions@oreilly.com).

For more information about our books, courses, conferences, and news, see our website at [\*http://www.oreilly.com\*](http://www.oreilly.com).

Find us on Facebook: [\*http://facebook.com/oreilly\*](http://facebook.com/oreilly)

Follow us on Twitter: [\*http://twitter.com/oreillymedia\*](http://twitter.com/oreillymedia)

Watch us on YouTube: [\*http://www.youtube.com/oreillymedia\*](http://www.youtube.com/oreillymedia)

## Acknowledgments

We would like to acknowledge the technical reviewers, Jake Vanderplas, Robert Kern, and Andrea Crotti, for their very helpful comments, as well as the general Python community for their support and encouragement. We would also like to thank the editors of the prior edition, Alex Martelli, Anna Ravenscroft, and David Ascher. Although this edition is newly written, the previous edition provided an initial framework for selecting the topics and recipes of interest. Last, but not least, we would like to thank readers of the early release editions for their comments and suggestions for improvement.

### David Beazley's Acknowledgments

Writing a book is no small task. As such, I would like to thank my wife Paula and my two boys for their patience and support during this project. Much of the material in this book was derived from content I developed teaching Python-related training classes over the last six years. Thus, I'd like to thank all of the students who have taken my courses and ultimately made this book possible. I'd also like to thank Ned Batchelder, Travis Oliphant, Peter Wang, Brian Van de Ven, Hugo Shi, Raymond Hettinger, Michael Foord, and Daniel Klein for traveling to the four corners of the world to teach these courses while I stayed home in Chicago to work on this project. Meghan Blanchette and Rachel Roumeliotis of O'Reilly were also instrumental in seeing this project through to completion despite the drama of several false starts and unforeseen delays. Last, but not least, I'd like to thank the Python community for their continued support and putting up with my flights of diabolical fancy.

David M. Beazley

[\*http://www.dabeaz.com\*](http://www.dabeaz.com)

[\*https://twitter.com/dabeaz\*](https://twitter.com/dabeaz)

## Brian Jones' Acknowledgments

I would like to thank both my coauthor, David Beazley, as well as Meghan Blanchette and Rachel Roumeliotis of O'Reilly, for working with me on this project. I would also like to thank my amazing wife, Natasha, for her patience and encouragement in this project, and her support in all of my ambitions. Most of all, I'd like to thank the Python community at large. Though I have contributed to the support of various open source projects, languages, clubs, and the like, no work has been so gratifying and rewarding as that which has been in the service of the Python community.

Brian K. Jones

*<http://www.protocolostomy.com>*

*<https://twitter.com/bkjones>*

---

# Data Structures and Algorithms

Python provides a variety of useful built-in data structures, such as lists, sets, and dictionaries. For the most part, the use of these structures is straightforward. However, common questions concerning searching, sorting, ordering, and filtering often arise. Thus, the goal of this chapter is to discuss common data structures and algorithms involving data. In addition, treatment is given to the various data structures contained in the collections module.

## 1.1. Unpacking a Sequence into Separate Variables

### Problem

You have an N-element tuple or sequence that you would like to unpack into a collection of N variables.

### Solution

Any sequence (or iterable) can be unpacked into variables using a simple assignment operation. The only requirement is that the number of variables and structure match the sequence. For example:

```
>>> p = (4, 5)
>>> x, y = p
>>> x
4
>>> y
5
>>>

>>> data = [ 'ACME', 50, 91.1, (2012, 12, 21) ]
>>> name, shares, price, date = data
>>> name
```

```

'ACME'
>>> date
(2012, 12, 21)

>>> name, shares, price, (year, mon, day) = data
>>> name
'ACME'
>>> year
2012
>>> mon
12
>>> day
21
>>>

```

If there is a mismatch in the number of elements, you'll get an error. For example:

```

>>> p = (4, 5)
>>> x, y, z = p
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: need more than 2 values to unpack
>>>

```

## Discussion

Unpacking actually works with any object that happens to be iterable, not just tuples or lists. This includes strings, files, iterators, and generators. For example:

```

>>> s = 'Hello'
>>> a, b, c, d, e = s
>>> a
'H'
>>> b
'e'
>>> e
'o'
>>>

```

When unpacking, you may sometimes want to discard certain values. Python has no special syntax for this, but you can often just pick a throwaway variable name for it. For example:

```

>>> data = [ 'ACME', 50, 91.1, (2012, 12, 21) ]
>>> _, shares, price, _ = data
>>> shares
50
>>> price
91.1
>>>

```

However, make sure that the variable name you pick isn't being used for something else already.

## 1.2. Unpacking Elements from Iterables of Arbitrary Length

### Problem

You need to unpack *N* elements from an iterable, but the iterable may be longer than *N* elements, causing a “too many values to unpack” exception.

### Solution

Python “star expressions” can be used to address this problem. For example, suppose you run a course and decide at the end of the semester that you’re going to drop the first and last homework grades, and only average the rest of them. If there are only four assignments, maybe you simply unpack all four, but what if there are 24? A star expression makes it easy:

```
def drop_first_last(grades):
    first, *middle, last = grades
    return avg(middle)
```

As another use case, suppose you have user records that consist of a name and email address, followed by an arbitrary number of phone numbers. You could unpack the records like this:

```
>>> record = ('Dave', 'dave@example.com', '773-555-1212', '847-555-1212')
>>> name, email, *phone_numbers = user_record
>>> name
'Dave'
>>> email
'dave@example.com'
>>> phone_numbers
['773-555-1212', '847-555-1212']
>>>
```

It’s worth noting that the `phone_numbers` variable will always be a list, regardless of how many phone numbers are unpacked (including none). Thus, any code that uses `phone_numbers` won’t have to account for the possibility that it might not be a list or perform any kind of additional type checking.

The starred variable can also be the first one in the list. For example, say you have a sequence of values representing your company’s sales figures for the last eight quarters. If you want to see how the most recent quarter stacks up to the average of the first seven, you could do something like this:

```
*trailing_qtrs, current_qtr = sales_record
trailing_avg = sum(trailing_qtrs) / len(trailing_qtrs)
return avg_comparison(trailing_avg, current_qtr)
```

Here’s a view of the operation from the Python interpreter:

```
>>> *trailing, current = [10, 8, 7, 1, 9, 5, 10, 3]
>>> trailing
[10, 8, 7, 1, 9, 5, 10]
>>> current
3
```

## Discussion

Extended iterable unpacking is tailor-made for unpacking iterables of unknown or arbitrary length. Oftentimes, these iterables have some known component or pattern in their construction (e.g. “everything after element 1 is a phone number”), and star unpacking lets the developer leverage those patterns easily instead of performing acrobatics to get at the relevant elements in the iterable.

It is worth noting that the star syntax can be especially useful when iterating over a sequence of tuples of varying length. For example, perhaps a sequence of tagged tuples:

```
records = [
    ('foo', 1, 2),
    ('bar', 'hello'),
    ('foo', 3, 4),
]

def do_foo(x, y):
    print('foo', x, y)

def do_bar(s):
    print('bar', s)

for tag, *args in records:
    if tag == 'foo':
        do_foo(*args)
    elif tag == 'bar':
        do_bar(*args)
```

Star unpacking can also be useful when combined with certain kinds of string processing operations, such as splitting. For example:

```
>>> line = 'nobody::-2:-2:Unprivileged User:/var/empty:/usr/bin/false'
>>> uname, *fields, homedir, sh = line.split(':')
>>> uname
'nobody'
>>> homedir
'/var/empty'
>>> sh
'/usr/bin/false'
>>>
```

Sometimes you might want to unpack values and throw them away. You can’t just specify a bare `*` when unpacking, but you could use a common throwaway variable name, such as `_` or `ign` (ignored). For example:

```
>>> record = ('ACME', 50, 123.45, (12, 18, 2012))
>>> name, *_ , (*_, year) = record
>>> name
'ACME'
>>> year
2012
>>>
```

There is a certain similarity between star unpacking and list-processing features of various functional languages. For example, if you have a list, you can easily split it into head and tail components like this:

```
>>> items = [1, 10, 7, 4, 5, 9]
>>> head, *tail = items
>>> head
1
>>> tail
[10, 7, 4, 5, 9]
>>>
```

One could imagine writing functions that perform such splitting in order to carry out some kind of clever recursive algorithm. For example:

```
>>> def sum(items):
...     head, *tail = items
...     return head + sum(tail) if tail else head
...
>>> sum(items)
36
>>>
```

However, be aware that recursion really isn't a strong Python feature due to the inherent recursion limit. Thus, this last example might be nothing more than an academic curiosity in practice.

## 1.3. Keeping the Last N Items

### Problem

You want to keep a limited history of the last few items seen during iteration or during some other kind of processing.

### Solution

Keeping a limited history is a perfect use for a `collections.deque`. For example, the following code performs a simple text match on a sequence of lines and yields the matching line along with the previous N lines of context when found:

```

from collections import deque

def search(lines, pattern, history=5):
    previous_lines = deque(maxlen=history)
    for line in lines:
        if pattern in line:
            yield line, previous_lines
        previous_lines.append(line)

# Example use on a file
if __name__ == '__main__':
    with open('somefile.txt') as f:
        for line, prevlines in search(f, 'python', 5):
            for pline in prevlines:
                print(pline, end='')
            print(line, end='')
            print('- '*20)

```

## Discussion

When writing code to search for items, it is common to use a generator function involving `yield`, as shown in this recipe's solution. This decouples the process of searching from the code that uses the results. If you're new to generators, see [Recipe 4.3](#).

Using `deque(maxlen=N)` creates a fixed-sized queue. When new items are added and the queue is full, the oldest item is automatically removed. For example:

```

>>> q = deque(maxlen=3)
>>> q.append(1)
>>> q.append(2)
>>> q.append(3)
>>> q
deque([1, 2, 3], maxlen=3)
>>> q.append(4)
>>> q
deque([2, 3, 4], maxlen=3)
>>> q.append(5)
>>> q
deque([3, 4, 5], maxlen=3)

```

Although you could manually perform such operations on a list (e.g., appending, deleting, etc.), the queue solution is far more elegant and runs a lot faster.

More generally, a deque can be used whenever you need a simple queue structure. If you don't give it a maximum size, you get an unbounded queue that lets you append and pop items on either end. For example:

```

>>> q = deque()
>>> q.append(1)
>>> q.append(2)
>>> q.append(3)
>>> q

```



```

deque([1, 2, 3])
>>> q.appendleft(4)
>>> q
deque([4, 1, 2, 3])
>>> q.pop()
3
>>> q
deque([4, 1, 2])
>>> q.popleft()
4

```

Adding or popping items from either end of a queue has  $O(1)$  complexity. This is unlike a list where inserting or removing items from the front of the list is  $O(N)$ .

## 1.4. Finding the Largest or Smallest N Items

### Problem

You want to make a list of the largest or smallest  $N$  items in a collection.

### Solution

The `heapq` module has two functions—`nlargest()` and `nsmallest()`—that do exactly what you want. For example:

```

import heapq

nums = [1, 8, 2, 23, 7, -4, 18, 23, 42, 37, 2]
print(heapq.nlargest(3, nums)) # Prints [42, 37, 23]
print(heapq.nsmallest(3, nums)) # Prints [-4, 1, 2]

```

Both functions also accept a key parameter that allows them to be used with more complicated data structures. For example:

```

portfolio = [
    {'name': 'IBM', 'shares': 100, 'price': 91.1},
    {'name': 'AAPL', 'shares': 50, 'price': 543.22},
    {'name': 'FB', 'shares': 200, 'price': 21.09},
    {'name': 'HPQ', 'shares': 35, 'price': 31.75},
    {'name': 'YHOO', 'shares': 45, 'price': 16.35},
    {'name': 'ACME', 'shares': 75, 'price': 115.65}
]

cheap = heapq.nsmallest(3, portfolio, key=lambda s: s['price'])
expensive = heapq.nlargest(3, portfolio, key=lambda s: s['price'])

```

### Discussion

If you are looking for the  $N$  smallest or largest items and  $N$  is small compared to the overall size of the collection, these functions provide superior performance. Underneath

the covers, they work by first converting the data into a list where items are ordered as a heap. For example:

```
>>> nums = [1, 8, 2, 23, 7, -4, 18, 23, 42, 37, 2]
>>> import heapq
>>> heap = list(nums)
>>> heapq.heapify(heap)
>>> heap
[-4, 2, 1, 23, 7, 2, 18, 23, 42, 37, 8]
>>>
```

The most important feature of a heap is that `heap[0]` is always the smallest item. Moreover, subsequent items can be easily found using the `heapq.heappop()` method, which pops off the first item and replaces it with the next smallest item (an operation that requires  $O(\log N)$  operations where  $N$  is the size of the heap). For example, to find the three smallest items, you would do this:

```
>>> heapq.heappop(heap)
-4
>>> heapq.heappop(heap)
1
>>> heapq.heappop(heap)
2
```

The `nlargest()` and `nsmallest()` functions are most appropriate if you are trying to find a relatively small number of items. If you are simply trying to find the single smallest or largest item ( $N=1$ ), it is faster to use `min()` and `max()`. Similarly, if  $N$  is about the same size as the collection itself, it is usually faster to sort it first and take a slice (i.e., use `sorted(items)[:N]` or `sorted(items)[-N:]`). It should be noted that the actual implementation of `nlargest()` and `nsmallest()` is adaptive in how it operates and will carry out some of these optimizations on your behalf (e.g., using sorting if  $N$  is close to the same size as the input).

Although it's not necessary to use this recipe, the implementation of a heap is an interesting and worthwhile subject of study. This can usually be found in any decent book on algorithms and data structures. The documentation for the `heapq` module also discusses the underlying implementation details.

## 1.5. Implementing a Priority Queue

### Problem

You want to implement a queue that sorts items by a given priority and always returns the item with the highest priority on each pop operation.

## Solution

The following class uses the `heapq` module to implement a simple priority queue:

```
import heapq

class PriorityQueue:
    def __init__(self):
        self._queue = []
        self._index = 0

    def push(self, item, priority):
        heapq.heappush(self._queue, (-priority, self._index, item))
        self._index += 1

    def pop(self):
        return heapq.heappop(self._queue)[-1]
```

Here is an example of how it might be used:

```
>>> class Item:
...     def __init__(self, name):
...         self.name = name
...     def __repr__(self):
...         return 'Item({!r})'.format(self.name)
...
>>> q = PriorityQueue()
>>> q.push(Item('foo'), 1)
>>> q.push(Item('bar'), 5)
>>> q.push(Item('spam'), 4)
>>> q.push(Item('grok'), 1)
>>> q.pop()
Item('bar')
>>> q.pop()
Item('spam')
>>> q.pop()
Item('foo')
>>> q.pop()
Item('grok')
>>>
```

Observe how the first `pop()` operation returned the item with the highest priority. Also observe how the two items with the same priority (foo and grok) were returned in the same order in which they were inserted into the queue.

## Discussion

The core of this recipe concerns the use of the `heapq` module. The functions `heapq.heappush()` and `heapq.heappop()` insert and remove items from a list `_queue` in a way such that the first item in the list has the smallest priority (as discussed in [Recipe 1.4](#)). The `heappop()` method always returns the “smallest” item, so that is the key to making the

queue pop the correct items. Moreover, since the push and pop operations have  $O(\log N)$  complexity where  $N$  is the number of items in the heap, they are fairly efficient even for fairly large values of  $N$ .

In this recipe, the queue consists of tuples of the form `(-priority, index, item)`. The `priority` value is negated to get the queue to sort items from highest priority to lowest priority. This is opposite of the normal heap ordering, which sorts from lowest to highest value.

The role of the `index` variable is to properly order items with the same priority level. By keeping a constantly increasing index, the items will be sorted according to the order in which they were inserted. However, the index also serves an important role in making the comparison operations work for items that have the same priority level.

To elaborate on that, instances of `Item` in the example can't be ordered. For example:

```
>>> a = Item('foo')
>>> b = Item('bar')
>>> a < b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: Item() < Item()
>>>
```

If you make `(priority, item)` tuples, they can be compared as long as the priorities are different. However, if two tuples with equal priorities are compared, the comparison fails as before. For example:

```
>>> a = (1, Item('foo'))
>>> b = (5, Item('bar'))
>>> a < b
True
>>> c = (1, Item('grok'))
>>> a < c
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: Item() < Item()
>>>
```

By introducing the extra index and making `(priority, index, item)` tuples, you avoid this problem entirely since no two tuples will ever have the same value for `index` (and Python never bothers to compare the remaining tuple values once the result of comparison can be determined):

```
>>> a = (1, 0, Item('foo'))
>>> b = (5, 1, Item('bar'))
>>> c = (1, 2, Item('grok'))
>>> a < b
True
>>> a < c
```

```
True
>>>
```

If you want to use this queue for communication between threads, you need to add appropriate locking and signaling. See [Recipe 12.3](#) for an example of how to do this.

The documentation for the `heapq` module has further examples and discussion concerning the theory and implementation of heaps.

## 1.6. Mapping Keys to Multiple Values in a Dictionary

### Problem

You want to make a dictionary that maps keys to more than one value (a so-called “multidict”).

### Solution

A dictionary is a mapping where each key is mapped to a single value. If you want to map keys to multiple values, you need to store the multiple values in another container such as a list or set. For example, you might make dictionaries like this:

```
d = {
    'a' : [1, 2, 3],
    'b' : [4, 5]
}

e = {
    'a' : {1, 2, 3},
    'b' : {4, 5}
}
```

The choice of whether or not to use lists or sets depends on intended use. Use a list if you want to preserve the insertion order of the items. Use a set if you want to eliminate duplicates (and don’t care about the order).

To easily construct such dictionaries, you can use `defaultdict` in the `collections` module. A feature of `defaultdict` is that it automatically initializes the first value so you can simply focus on adding items. For example:

```
from collections import defaultdict

d = defaultdict(list)
d['a'].append(1)
d['a'].append(2)
d['b'].append(4)
...

d = defaultdict(set)
```

```

d['a'].add(1)
d['a'].add(2)
d['b'].add(4)
...

```

One caution with `defaultdict` is that it will automatically create dictionary entries for keys accessed later on (even if they aren't currently found in the dictionary). If you don't want this behavior, you might use `setdefault()` on an ordinary dictionary instead. For example:

```

d = {}      # A regular dictionary
d.setdefault('a', []).append(1)
d.setdefault('a', []).append(2)
d.setdefault('b', []).append(4)
...

```

However, many programmers find `setdefault()` to be a little unnatural—not to mention the fact that it always creates a new instance of the initial value on each invocation (the empty list `[]` in the example).

## Discussion

In principle, constructing a multivalued dictionary is simple. However, initialization of the first value can be messy if you try to do it yourself. For example, you might have code that looks like this:

```

d = {}
for key, value in pairs:
    if key not in d:
        d[key] = []
    d[key].append(value)

```

Using a `defaultdict` simply leads to much cleaner code:

```

d = defaultdict(list)
for key, value in pairs:
    d[key].append(value)

```

This recipe is strongly related to the problem of grouping records together in data processing problems. See [Recipe 1.15](#) for an example.

## 1.7. Keeping Dictionaries in Order

### Problem

You want to create a dictionary, and you also want to control the order of items when iterating or serializing.

## Solution

To control the order of items in a dictionary, you can use an `OrderedDict` from the `collections` module. It exactly preserves the original insertion order of data when iterating. For example:

```
from collections import OrderedDict

d = OrderedDict()
d['foo'] = 1
d['bar'] = 2
d['spam'] = 3
d['grok'] = 4

# Outputs "foo 1", "bar 2", "spam 3", "grok 4"
for key in d:
    print(key, d[key])
```

An `OrderedDict` can be particularly useful when you want to build a mapping that you may want to later serialize or encode into a different format. For example, if you want to precisely control the order of fields appearing in a JSON encoding, first building the data in an `OrderedDict` will do the trick:

```
>>> import json
>>> json.dumps(d)
'{"foo": 1, "bar": 2, "spam": 3, "grok": 4}'
>>>
```

## Discussion

An `OrderedDict` internally maintains a doubly linked list that orders the keys according to insertion order. When a new item is first inserted, it is placed at the end of this list. Subsequent reassignment of an existing key doesn't change the order.

Be aware that the size of an `OrderedDict` is more than twice as large as a normal dictionary due to the extra linked list that's created. Thus, if you are going to build a data structure involving a large number of `OrderedDict` instances (e.g., reading 100,000 lines of a CSV file into a list of `OrderedDict` instances), you would need to study the requirements of your application to determine if the benefits of using an `OrderedDict` outweighed the extra memory overhead.

## 1.8. Calculating with Dictionaries

### Problem

You want to perform various calculations (e.g., minimum value, maximum value, sorting, etc.) on a dictionary of data.

## Solution

Consider a dictionary that maps stock names to prices:

```
prices = {
    'ACME': 45.23,
    'AAPL': 612.78,
    'IBM': 205.55,
    'HPQ': 37.20,
    'FB': 10.75
}
```

In order to perform useful calculations on the dictionary contents, it is often useful to invert the keys and values of the dictionary using `zip()`. For example, here is how to find the minimum and maximum price and stock name:

```
min_price = min(zip(prices.values(), prices.keys()))
# min_price is (10.75, 'FB')

max_price = max(zip(prices.values(), prices.keys()))
# max_price is (612.78, 'AAPL')
```

Similarly, to rank the data, use `zip()` with `sorted()`, as in the following:

```
prices_sorted = sorted(zip(prices.values(), prices.keys()))
# prices_sorted is [(10.75, 'FB'), (37.2, 'HPQ'),
#                  (45.23, 'ACME'), (205.55, 'IBM'),
#                  (612.78, 'AAPL')]
```

When doing these calculations, be aware that `zip()` creates an iterator that can only be consumed once. For example, the following code is an error:

```
prices_and_names = zip(prices.values(), prices.keys())
print(min(prices_and_names)) # OK
print(max(prices_and_names)) # ValueError: max() arg is an empty sequence
```

## Discussion

If you try to perform common data reductions on a dictionary, you'll find that they only process the keys, not the values. For example:

```
min(prices) # Returns 'AAPL'
max(prices) # Returns 'IBM'
```

This is probably not what you want because you're actually trying to perform a calculation involving the dictionary values. You might try to fix this using the `values()` method of a dictionary:

```
min(prices.values()) # Returns 10.75
max(prices.values()) # Returns 612.78
```



Unfortunately, this is often not exactly what you want either. For example, you may want to know information about the corresponding keys (e.g., which stock has the lowest price?).

You can get the key corresponding to the min or max value if you supply a key function to `min()` and `max()`. For example:

```
min(prices, key=lambda k: prices[k]) # Returns 'FB'
max(prices, key=lambda k: prices[k]) # Returns 'AAPL'
```

However, to get the minimum value, you'll need to perform an extra lookup step. For example:

```
min_value = prices[min(prices, key=lambda k: prices[k])]
```

The solution involving `zip()` solves the problem by “inverting” the dictionary into a sequence of (value, key) pairs. When performing comparisons on such tuples, the value element is compared first, followed by the key. This gives you exactly the behavior that you want and allows reductions and sorting to be easily performed on the dictionary contents using a single statement.

It should be noted that in calculations involving (value, key) pairs, the key will be used to determine the result in instances where multiple entries happen to have the same value. For instance, in calculations such as `min()` and `max()`, the entry with the smallest or largest key will be returned if there happen to be duplicate values. For example:

```
>>> prices = { 'AAA' : 45.23, 'ZZZ' : 45.23 }
>>> min(zip(prices.values(), prices.keys()))
(45.23, 'AAA')
>>> max(zip(prices.values(), prices.keys()))
(45.23, 'ZZZ')
>>>
```

## 1.9. Finding Commonalities in Two Dictionaries

### Problem

You have two dictionaries and want to find out what they might have in common (same keys, same values, etc.).

### Solution

Consider two dictionaries:

```
a = {
    'x' : 1,
    'y' : 2,
    'z' : 3
}
```

```
b = {
    'w' : 10,
    'x' : 11,
    'y' : 2
}
```

To find out what the two dictionaries have in common, simply perform common set operations using the `keys()` or `items()` methods. For example:

```
# Find keys in common
a.keys() & b.keys() # { 'x', 'y' }

# Find keys in a that are not in b
a.keys() - b.keys() # { 'z' }

# Find (key,value) pairs in common
a.items() & b.items() # { ('y', 2) }
```

These kinds of operations can also be used to alter or filter dictionary contents. For example, suppose you want to make a new dictionary with selected keys removed. Here is some sample code using a dictionary comprehension:

```
# Make a new dictionary with certain keys removed
c = {key:a[key] for key in a.keys() - {'z', 'w'}}
# c is {'x': 1, 'y': 2}
```

## Discussion

A dictionary is a mapping between a set of keys and values. The `keys()` method of a dictionary returns a keys-view object that exposes the keys. A little-known feature of keys views is that they also support common set operations such as unions, intersections, and differences. Thus, if you need to perform common set operations with dictionary keys, you can often just use the keys-view objects directly without first converting them into a set.

The `items()` method of a dictionary returns an items-view object consisting of (key, value) pairs. This object supports similar set operations and can be used to perform operations such as finding out which key-value pairs two dictionaries have in common.

Although similar, the `values()` method of a dictionary does not support the set operations described in this recipe. In part, this is due to the fact that unlike keys, the items contained in a values view aren't guaranteed to be unique. This alone makes certain set operations of questionable utility. However, if you must perform such calculations, they can be accomplished by simply converting the values to a set first.

# 1.10. Removing Duplicates from a Sequence while Maintaining Order

## Problem

You want to eliminate the duplicate values in a sequence, but preserve the order of the remaining items.

## Solution

If the values in the sequence are hashable, the problem can be easily solved using a set and a generator. For example:

```
def dedupe(items):
    seen = set()
    for item in items:
        if item not in seen:
            yield item
            seen.add(item)
```

Here is an example of how to use your function:

```
>>> a = [1, 5, 2, 1, 9, 1, 5, 10]
>>> list(dedupe(a))
[1, 5, 2, 9, 10]
>>>
```

This only works if the items in the sequence are hashable. If you are trying to eliminate duplicates in a sequence of unhashable types (such as dicts), you can make a slight change to this recipe, as follows:

```
def dedupe(items, key=None):
    seen = set()
    for item in items:
        val = item if key is None else key(item)
        if val not in seen:
            yield item
            seen.add(val)
```

Here, the purpose of the key argument is to specify a function that converts sequence items into a hashable type for the purposes of duplicate detection. Here's how it works:

```
>>> a = [{'x':1, 'y':2}, {'x':1, 'y':3}, {'x':1, 'y':2}, {'x':2, 'y':4}]
>>> list(dedupe(a, key=lambda d: (d['x'],d['y'])))
[{'x': 1, 'y': 2}, {'x': 1, 'y': 3}, {'x': 2, 'y': 4}]
>>> list(dedupe(a, key=lambda d: d['x']))
[{'x': 1, 'y': 2}, {'x': 2, 'y': 4}]
>>>
```

This latter solution also works nicely if you want to eliminate duplicates based on the value of a single field or attribute or a larger data structure.

## Discussion

If all you want to do is eliminate duplicates, it is often easy enough to make a set. For example:

```
>>> a
[1, 5, 2, 1, 9, 1, 5, 10]
>>> set(a)
{1, 2, 10, 5, 9}
>>>
```

However, this approach doesn't preserve any kind of ordering. So, the resulting data will be scrambled afterward. The solution shown avoids this.

The use of a generator function in this recipe reflects the fact that you might want the function to be extremely general purpose—not necessarily tied directly to list processing. For example, if you want to read a file, eliminating duplicate lines, you could simply do this:

```
with open(somefile, 'r') as f:
    for line in dedupe(f):
        ...
```

The specification of a key function mimics similar functionality in built-in functions such as `sorted()`, `min()`, and `max()`. For instance, see Recipes 1.8 and 1.13.

## 1.11. Naming a Slice

### Problem

Your program has become an unreadable mess of hardcoded slice indices and you want to clean it up.

### Solution

Suppose you have some code that is pulling specific data fields out of a record string with fixed fields (e.g., from a flat file or similar format):

```
##### 0123456789012345678901234567890123456789012345678901234567890'
record = '.....100.....513.25.....'
cost = int(record[20:32]) * float(record[40:48])
```

Instead of doing that, why not name the slices like this?

```
SHARES = slice(20,32)
PRICE = slice(40,48)

cost = int(record[SHARES]) * float(record[PRICE])
```

In the latter version, you avoid having a lot of mysterious hardcoded indices, and what you're doing becomes much clearer.

## Discussion

As a general rule, writing code with a lot of hardcoded index values leads to a readability and maintenance mess. For example, if you come back to the code a year later, you'll look at it and wonder what you were thinking when you wrote it. The solution shown is simply a way of more clearly stating what your code is actually doing.

In general, the built-in `slice()` creates a slice object that can be used anywhere a slice is allowed. For example:

```
>>> items = [0, 1, 2, 3, 4, 5, 6]
>>> a = slice(2, 4)
>>> items[2:4]
[2, 3]
>>> items[a]
[2, 3]
>>> items[a] = [10, 11]
>>> items
[0, 1, 10, 11, 4, 5, 6]
>>> del items[a]
>>> items
[0, 1, 4, 5, 6]
```

If you have a slice instance `s`, you can get more information about it by looking at its `s.start`, `s.stop`, and `s.step` attributes, respectively. For example:

```
>>> a = slice(10, 50, 2)
>>> a.start
10
>>> a.stop
50
>>> a.step
2
>>>
```

In addition, you can map a slice onto a sequence of a specific size by using its `indices(size)` method. This returns a tuple (`start`, `stop`, `step`) where all values have been suitably limited to fit within bounds (as to avoid `IndexError` exceptions when indexing). For example:

```
>>> s = 'HelloWorld'
>>> a.indices(len(s))
(5, 10, 2)
>>> for i in range(*a.indices(len(s))):
...     print(s[i])
...
W
r
```

```
d
>>>
```

## 1.12. Determining the Most Frequently Occurring Items in a Sequence

### Problem

You have a sequence of items, and you'd like to determine the most frequently occurring items in the sequence.

### Solution

The `collections.Counter` class is designed for just such a problem. It even comes with a handy `most_common()` method that will give you the answer.

To illustrate, let's say you have a list of words and you want to find out which words occur most often. Here's how you would do it:

```
words = [
    'look', 'into', 'my', 'eyes', 'look', 'into', 'my', 'eyes',
    'the', 'eyes', 'the', 'eyes', 'the', 'eyes', 'not', 'around', 'the',
    'eyes', "don't", 'look', 'around', 'the', 'eyes', 'look', 'into',
    'my', 'eyes', "you're", 'under'
]

from collections import Counter
word_counts = Counter(words)
top_three = word_counts.most_common(3)
print(top_three)
# Outputs [('eyes', 8), ('the', 5), ('look', 4)]
```

### Discussion

As input, `Counter` objects can be fed any sequence of hashable input items. Under the covers, a `Counter` is a dictionary that maps the items to the number of occurrences. For example:

```
>>> word_counts['not']
1
>>> word_counts['eyes']
8
>>>
```

If you want to increment the count manually, simply use addition:

```
>>> morewords = ['why', 'are', 'you', 'not', 'looking', 'in', 'my', 'eyes']
>>> for word in morewords:
...     word_counts[word] += 1
```

```
...
>>> word_counts['eyes']
9
>>>
```

Or, alternatively, you could use the `update()` method:

```
>>> word_counts.update(morewords)
>>>
```

A little-known feature of `Counter` instances is that they can be easily combined using various mathematical operations. For example:

```
>>> a = Counter(words)
>>> b = Counter(morewords)
>>> a
Counter({'eyes': 8, 'the': 5, 'look': 4, 'into': 3, 'my': 3, 'around': 2,
        'you're': 1, 'don't': 1, 'under': 1, 'not': 1})
>>> b
Counter({'eyes': 1, 'looking': 1, 'are': 1, 'in': 1, 'not': 1, 'you': 1,
        'my': 1, 'why': 1})

>>> # Combine counts
>>> c = a + b
>>> c
Counter({'eyes': 9, 'the': 5, 'look': 4, 'my': 4, 'into': 3, 'not': 2,
        'around': 2, 'you're': 1, 'don't': 1, 'in': 1, 'why': 1,
        'looking': 1, 'are': 1, 'under': 1, 'you': 1})

>>> # Subtract counts
>>> d = a - b
>>> d
Counter({'eyes': 7, 'the': 5, 'look': 4, 'into': 3, 'my': 2, 'around': 2,
        'you're': 1, 'don't': 1, 'under': 1})
>>>
```

Needless to say, `Counter` objects are a tremendously useful tool for almost any kind of problem where you need to tabulate and count data. You should prefer this over manually written solutions involving dictionaries.

## 1.13. Sorting a List of Dictionaries by a Common Key

### Problem

You have a list of dictionaries and you would like to sort the entries according to one or more of the dictionary values.

## Solution

Sorting this type of structure is easy using the `operator` module's `itemgetter` function. Let's say you've queried a database table to get a listing of the members on your website, and you receive the following data structure in return:

```
rows = [
    {'fname': 'Brian', 'lname': 'Jones', 'uid': 1003},
    {'fname': 'David', 'lname': 'Beazley', 'uid': 1002},
    {'fname': 'John', 'lname': 'Cleese', 'uid': 1001},
    {'fname': 'Big', 'lname': 'Jones', 'uid': 1004}
]
```

It's fairly easy to output these rows ordered by any of the fields common to all of the dictionaries. For example:

```
from operator import itemgetter

rows_by_fname = sorted(rows, key=itemgetter('fname'))
rows_by_uid = sorted(rows, key=itemgetter('uid'))

print(rows_by_fname)
print(rows_by_uid)
```

The preceding code would output the following:

```
[{'fname': 'Big', 'uid': 1004, 'lname': 'Jones'},
 {'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'},
 {'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},
 {'fname': 'John', 'uid': 1001, 'lname': 'Cleese'}]

[{'fname': 'John', 'uid': 1001, 'lname': 'Cleese'},
 {'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},
 {'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'},
 {'fname': 'Big', 'uid': 1004, 'lname': 'Jones'}]
```

The `itemgetter()` function can also accept multiple keys. For example, this code

```
rows_by_lfname = sorted(rows, key=itemgetter('lname', 'fname'))
print(rows_by_lfname)
```

Produces output like this:

```
[{'fname': 'David', 'uid': 1002, 'lname': 'Beazley'},
 {'fname': 'John', 'uid': 1001, 'lname': 'Cleese'},
 {'fname': 'Big', 'uid': 1004, 'lname': 'Jones'},
 {'fname': 'Brian', 'uid': 1003, 'lname': 'Jones'}]
```

## Discussion

In this example, `rows` is passed to the built-in `sorted()` function, which accepts a key-word argument `key`. This argument is expected to be a callable that accepts a single item



from `rows` as input and returns a value that will be used as the basis for sorting. The `itemgetter()` function creates just such a callable.

The operator `itemgetter()` function takes as arguments the lookup indices used to extract the desired values from the records in `rows`. It can be a dictionary key name, a numeric list element, or any value that can be fed to an object's `__getitem__()` method. If you give multiple indices to `itemgetter()`, the callable it produces will return a tuple with all of the elements in it, and `sorted()` will order the output according to the sorted order of the tuples. This can be useful if you want to simultaneously sort on multiple fields (such as last and first name, as shown in the example).

The functionality of `itemgetter()` is sometimes replaced by `lambda` expressions. For example:

```
rows_by_fname = sorted(rows, key=lambda r: r['fname'])
rows_by_lfname = sorted(rows, key=lambda r: (r['lname'], r['fname']))
```

This solution often works just fine. However, the solution involving `itemgetter()` typically runs a bit faster. Thus, you might prefer it if performance is a concern.

Last, but not least, don't forget that the technique shown in this recipe can be applied to functions such as `min()` and `max()`. For example:

```
>>> min(rows, key=itemgetter('uid'))
{'fname': 'John', 'lname': 'Cleese', 'uid': 1001}
>>> max(rows, key=itemgetter('uid'))
{'fname': 'Big', 'lname': 'Jones', 'uid': 1004}
>>>
```

## 1.14. Sorting Objects Without Native Comparison Support

### Problem

You want to sort objects of the same class, but they don't natively support comparison operations.

### Solution

The built-in `sorted()` function takes a key argument that can be passed a callable that will return some value in the object that `sorted` will use to compare the objects. For example, if you have a sequence of `User` instances in your application, and you want to sort them by their `user_id` attribute, you would supply a callable that takes a `User` instance as input and returns the `user_id`. For example:

```
>>> class User:
...     def __init__(self, user_id):
...         self.user_id = user_id
```

```

...     def __repr__(self):
...         return 'User({})'.format(self.user_id)
...
>>> users = [User(23), User(3), User(99)]
>>> users
[User(23), User(3), User(99)]
>>> sorted(users, key=lambda u: u.user_id)
[User(3), User(23), User(99)]
>>>

```

Instead of using `lambda`, an alternative approach is to use `operator.attrgetter()`:

```

>>> from operator import attrgetter
>>> sorted(users, key=attrgetter('user_id'))
[User(3), User(23), User(99)]
>>>

```

## Discussion

The choice of whether or not to use `lambda` or `attrgetter()` may be one of personal preference. However, `attrgetter()` is often a tad bit faster and also has the added feature of allowing multiple fields to be extracted simultaneously. This is analogous to the use of `operator.itemgetter()` for dictionaries (see [Recipe 1.13](#)). For example, if `User` instances also had a `first_name` and `last_name` attribute, you could perform a sort like this:

```
by_name = sorted(users, key=attrgetter('last_name', 'first_name'))
```

It is also worth noting that the technique used in this recipe can be applied to functions such as `min()` and `max()`. For example:

```

>>> min(users, key=attrgetter('user_id'))
User(3)
>>> max(users, key=attrgetter('user_id'))
User(99)
>>>

```

## 1.15. Grouping Records Together Based on a Field

### Problem

You have a sequence of dictionaries or instances and you want to iterate over the data in groups based on the value of a particular field, such as `date`.

### Solution

The `itertools.groupby()` function is particularly useful for grouping data together like this. To illustrate, suppose you have the following list of dictionaries:

```
rows = [
    {'address': '5412 N CLARK', 'date': '07/01/2012'},
    {'address': '5148 N CLARK', 'date': '07/04/2012'},
    {'address': '5800 E 58TH', 'date': '07/02/2012'},
    {'address': '2122 N CLARK', 'date': '07/03/2012'},
    {'address': '5645 N RAVENSWOOD', 'date': '07/02/2012'},
    {'address': '1060 W ADDISON', 'date': '07/02/2012'},
    {'address': '4801 N BROADWAY', 'date': '07/01/2012'},
    {'address': '1039 W GRANVILLE', 'date': '07/04/2012'},
]
```

Now suppose you want to iterate over the data in chunks grouped by date. To do it, first sort by the desired field (in this case, date) and then use `itertools.groupby()`:

```
from operator import itemgetter
from itertools import groupby

# Sort by the desired field first
rows.sort(key=itemgetter('date'))

# Iterate in groups
for date, items in groupby(rows, key=itemgetter('date')):
    print(date)
    for i in items:
        print(' ', i)
```

This produces the following output:

```
07/01/2012
    {'date': '07/01/2012', 'address': '5412 N CLARK'}
    {'date': '07/01/2012', 'address': '4801 N BROADWAY'}
07/02/2012
    {'date': '07/02/2012', 'address': '5800 E 58TH'}
    {'date': '07/02/2012', 'address': '5645 N RAVENSWOOD'}
    {'date': '07/02/2012', 'address': '1060 W ADDISON'}
07/03/2012
    {'date': '07/03/2012', 'address': '2122 N CLARK'}
07/04/2012
    {'date': '07/04/2012', 'address': '5148 N CLARK'}
    {'date': '07/04/2012', 'address': '1039 W GRANVILLE'}
```

## Discussion

The `groupby()` function works by scanning a sequence and finding sequential “runs” of identical values (or values returned by the given key function). On each iteration, it returns the value along with an iterator that produces all of the items in a group with the same value.

An important preliminary step is sorting the data according to the field of interest. Since `groupby()` only examines consecutive items, failing to sort first won’t group the records as you want.

If your goal is to simply group the data together by dates into a large data structure that allows random access, you may have better luck using `defaultdict()` to build a `multidict`, as described in [Recipe 1.6](#). For example:

```
from collections import defaultdict
rows_by_date = defaultdict(list)
for row in rows:
    rows_by_date[row['date']].append(row)
```

This allows the records for each date to be accessed easily like this:

```
>>> for r in rows_by_date['07/01/2012']:
...     print(r)
...
{'date': '07/01/2012', 'address': '5412 N CLARK'}
{'date': '07/01/2012', 'address': '4801 N BROADWAY'}
>>>
```

For this latter example, it's not necessary to sort the records first. Thus, if memory is no concern, it may be faster to do this than to first sort the records and iterate using `groupby()`.

## 1.16. Filtering Sequence Elements

### Problem

You have data inside of a sequence, and need to extract values or reduce the sequence using some criteria.

### Solution

The easiest way to filter sequence data is often to use a list comprehension. For example:

```
>>> mylist = [1, 4, -5, 10, -7, 2, 3, -1]
>>> [n for n in mylist if n > 0]
[1, 4, 10, 2, 3]
>>> [n for n in mylist if n < 0]
[-5, -7, -1]
>>>
```

One potential downside of using a list comprehension is that it might produce a large result if the original input is large. If this is a concern, you can use generator expressions to produce the filtered values iteratively. For example:

```
>>> pos = (n for n in mylist if n > 0)
>>> pos
<generator object <genexpr> at 0x1006a0eb0>
>>> for x in pos:
...     print(x)
...
...
```

```

1
4
10
2
3
>>>

```

Sometimes, the filtering criteria cannot be easily expressed in a list comprehension or generator expression. For example, suppose that the filtering process involves exception handling or some other complicated detail. For this, put the filtering code into its own function and use the built-in `filter()` function. For example:

```

values = ['1', '2', '-3', '-', '4', 'N/A', '5']

def is_int(val):
    try:
        x = int(val)
        return True
    except ValueError:
        return False

ivals = list(filter(is_int, values))
print(ivals)
# Outputs ['1', '2', '-3', '4', '5']

```

`filter()` creates an iterator, so if you want to create a list of results, make sure you also use `list()` as shown.

## Discussion

List comprehensions and generator expressions are often the easiest and most straightforward ways to filter simple data. They also have the added power to transform the data at the same time. For example:

```

>>> mylist = [1, 4, -5, 10, -7, 2, 3, -1]
>>> import math
>>> [math.sqrt(n) for n in mylist if n > 0]
[1.0, 2.0, 3.1622776601683795, 1.4142135623730951, 1.7320508075688772]
>>>

```

One variation on filtering involves replacing the values that don't meet the criteria with a new value instead of discarding them. For example, perhaps instead of just finding positive values, you want to also clip bad values to fit within a specified range. This is often easily accomplished by moving the filter criterion into a conditional expression like this:

```

>>> clip_neg = [n if n > 0 else 0 for n in mylist]
>>> clip_neg
[1, 4, 0, 10, 0, 2, 3, 0]
>>> clip_pos = [n if n < 0 else 0 for n in mylist]
>>> clip_pos

```

```
[0, 0, -5, 0, -7, 0, 0, -1]
>>>
```

Another notable filtering tool is `itertools.compress()`, which takes an iterable and an accompanying Boolean selector sequence as input. As output, it gives you all of the items in the iterable where the corresponding element in the selector is `True`. This can be useful if you're trying to apply the results of filtering one sequence to another related sequence. For example, suppose you have the following two columns of data:

```
addresses = [
    '5412 N CLARK',
    '5148 N CLARK',
    '5800 E 58TH',
    '2122 N CLARK',
    '5645 N RAVENSWOOD',
    '1060 W ADDISON',
    '4801 N BROADWAY',
    '1039 W GRANVILLE',
]

counts = [ 0, 3, 10, 4, 1, 7, 6, 1]
```

Now suppose you want to make a list of all addresses where the corresponding count value was greater than 5. Here's how you could do it:

```
>>> from itertools import compress
>>> more5 = [n > 5 for n in counts]
>>> more5
[False, False, True, False, False, True, True, False]
>>> list(compress(addresses, more5))
['5800 E 58TH', '4801 N BROADWAY', '1039 W GRANVILLE']
>>>
```

The key here is to first create a sequence of Booleans that indicates which elements satisfy the desired condition. The `compress()` function then picks out the items corresponding to `True` values.

Like `filter()`, `compress()` normally returns an iterator. Thus, you need to use `list()` to turn the results into a list if desired.

## 1.17. Extracting a Subset of a Dictionary

### Problem

You want to make a dictionary that is a subset of another dictionary.

## Solution

This is easily accomplished using a dictionary comprehension. For example:

```
prices = {
    'ACME': 45.23,
    'AAPL': 612.78,
    'IBM': 205.55,
    'HPQ': 37.20,
    'FB': 10.75
}

# Make a dictionary of all prices over 200
p1 = { key:value for key, value in prices.items() if value > 200 }

# Make a dictionary of tech stocks
tech_names = { 'AAPL', 'IBM', 'HPQ', 'MSFT' }
p2 = { key:value for key,value in prices.items() if key in tech_names }
```

## Discussion

Much of what can be accomplished with a dictionary comprehension might also be done by creating a sequence of tuples and passing them to the `dict()` function. For example:

```
p1 = dict((key, value) for key, value in prices.items() if value > 200)
```

However, the dictionary comprehension solution is a bit clearer and actually runs quite a bit faster (over twice as fast when tested on the `prices` dictionary used in the example).

Sometimes there are multiple ways of accomplishing the same thing. For instance, the second example could be rewritten as:

```
# Make a dictionary of tech stocks
tech_names = { 'AAPL', 'IBM', 'HPQ', 'MSFT' }
p2 = { key:prices[key] for key in prices.keys() & tech_names }
```

However, a timing study reveals that this solution is almost 1.6 times slower than the first solution. If performance matters, it usually pays to spend a bit of time studying it. See [Recipe 14.13](#) for specific information about timing and profiling.

## 1.18. Mapping Names to Sequence Elements

### Problem

You have code that accesses list or tuple elements by position, but this makes the code somewhat difficult to read at times. You'd also like to be less dependent on position in the structure, by accessing the elements by name.

## Solution

`collections.namedtuple()` provides these benefits, while adding minimal overhead over using a normal tuple object. `collections.namedtuple()` is actually a factory method that returns a subclass of the standard Python tuple type. You feed it a type name, and the fields it should have, and it returns a class that you can instantiate, passing in values for the fields you've defined, and so on. For example:

```
>>> from collections import namedtuple
>>> Subscriber = namedtuple('Subscriber', ['addr', 'joined'])
>>> sub = Subscriber('jonesy@example.com', '2012-10-19')
>>> sub
Subscriber(addr='jonesy@example.com', joined='2012-10-19')
>>> sub.addr
'jonesy@example.com'
>>> sub.joined
'2012-10-19'
>>>
```

Although an instance of a `namedtuple` looks like a normal class instance, it is interchangeable with a tuple and supports all of the usual tuple operations such as indexing and unpacking. For example:

```
>>> len(sub)
2
>>> addr, joined = sub
>>> addr
'jonesy@example.com'
>>> joined
'2012-10-19'
>>>
```

A major use case for named tuples is decoupling your code from the position of the elements it manipulates. So, if you get back a large list of tuples from a database call, then manipulate them by accessing the positional elements, your code could break if, say, you added a new column to your table. Not so if you first cast the returned tuples to `namedtuples`.

To illustrate, here is some code using ordinary tuples:

```
def compute_cost(records):
    total = 0.0
    for rec in records:
        total += rec[1] * rec[2]
    return total
```

References to positional elements often make the code a bit less expressive and more dependent on the structure of the records. Here is a version that uses a `namedtuple`:

```
from collections import namedtuple

Stock = namedtuple('Stock', ['name', 'shares', 'price'])
```



```
def compute_cost(records):
    total = 0.0
    for rec in records:
        s = Stock(*rec)
        total += s.shares * s.price
    return total
```

Naturally, you can avoid the explicit conversion to the `Stock` namedtuple if the records sequence in the example already contained such instances.

## Discussion

One possible use of a namedtuple is as a replacement for a dictionary, which requires more space to store. Thus, if you are building large data structures involving dictionaries, use of a namedtuple will be more efficient. However, be aware that unlike a dictionary, a namedtuple is immutable. For example:

```
>>> s = Stock('ACME', 100, 123.45)
>>> s
Stock(name='ACME', shares=100, price=123.45)
>>> s.shares = 75
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>
```

If you need to change any of the attributes, it can be done using the `_replace()` method of a namedtuple instance, which makes an entirely new namedtuple with specified values replaced. For example:

```
>>> s = s._replace(shares=75)
>>> s
Stock(name='ACME', shares=75, price=123.45)
>>>
```

A subtle use of the `_replace()` method is that it can be a convenient way to populate named tuples that have optional or missing fields. To do this, you make a prototype tuple containing the default values and then use `_replace()` to create new instances with values replaced. For example:

```
from collections import namedtuple

Stock = namedtuple('Stock', ['name', 'shares', 'price', 'date', 'time'])

# Create a prototype instance
stock_prototype = Stock('', 0, 0.0, None, None)

# Function to convert a dictionary to a Stock
def dict_to_stock(s):
    return stock_prototype._replace(**s)
```

Here is an example of how this code would work:

```
>>> a = {'name': 'ACME', 'shares': 100, 'price': 123.45}
>>> dict_to_stock(a)
Stock(name='ACME', shares=100, price=123.45, date=None, time=None)
>>> b = {'name': 'ACME', 'shares': 100, 'price': 123.45, 'date': '12/17/2012'}
>>> dict_to_stock(b)
Stock(name='ACME', shares=100, price=123.45, date='12/17/2012', time=None)
>>>
```

Last, but not least, it should be noted that if your goal is to define an efficient data structure where you will be changing various instance attributes, using `namedtuple` is not your best choice. Instead, consider defining a class using `__slots__` instead (see [Recipe 8.4](#)).

## 1.19. Transforming and Reducing Data at the Same Time

### Problem

You need to execute a reduction function (e.g., `sum()`, `min()`, `max()`), but first need to transform or filter the data.

### Solution

A very elegant way to combine a data reduction and a transformation is to use a generator-expression argument. For example, if you want to calculate the sum of squares, do the following:

```
nums = [1, 2, 3, 4, 5]
s = sum(x * x for x in nums)
```

Here are a few other examples:

```
# Determine if any .py files exist in a directory
import os
files = os.listdir('dirname')
if any(name.endswith('.py') for name in files):
    print('There be python!')
else:
    print('Sorry, no python.')
```

```
# Output a tuple as CSV
s = ('ACME', 50, 123.45)
print(','.join(str(x) for x in s))
```

```
# Data reduction across fields of a data structure
portfolio = [
    {'name': 'GOOG', 'shares': 50},
    {'name': 'YHOO', 'shares': 75},
    {'name': 'AOL', 'shares': 20},
```

```

    {'name': 'SCOX', 'shares': 65}
]
min_shares = min(s['shares'] for s in portfolio)

```

## Discussion

The solution shows a subtle syntactic aspect of generator expressions when supplied as the single argument to a function (i.e., you don't need repeated parentheses). For example, these statements are the same:

```

s = sum((x * x for x in nums))    # Pass generator-expr as argument
s = sum(x * x for x in nums)     # More elegant syntax

```

Using a generator argument is often a more efficient and elegant approach than first creating a temporary list. For example, if you didn't use a generator expression, you might consider this alternative implementation:

```

nums = [1, 2, 3, 4, 5]
s = sum([x * x for x in nums])

```

This works, but it introduces an extra step and creates an extra list. For such a small list, it might not matter, but if `nums` was huge, you would end up creating a large temporary data structure to only be used once and discarded. The generator solution transforms the data iteratively and is therefore much more memory-efficient.

Certain reduction functions such as `min()` and `max()` accept a key argument that might be useful in situations where you might be inclined to use a generator. For example, in the `portfolio` example, you might consider this alternative:

```

# Original: Returns 20
min_shares = min(s['shares'] for s in portfolio)

# Alternative: Returns {'name': 'AOL', 'shares': 20}
min_shares = min(portfolio, key=lambda s: s['shares'])

```

## 1.20. Combining Multiple Mappings into a Single Mapping

### Problem

You have multiple dictionaries or mappings that you want to logically combine into a single mapping to perform certain operations, such as looking up values or checking for the existence of keys.

### Solution

Suppose you have two dictionaries:

```
a = {'x': 1, 'z': 3 }
b = {'y': 2, 'z': 4 }
```

Now suppose you want to perform lookups where you have to check both dictionaries (e.g., first checking in `a` and then in `b` if not found). An easy way to do this is to use the `ChainMap` class from the `collections` module. For example:

```
from collections import ChainMap
c = ChainMap(a,b)
print(c['x'])    # Outputs 1 (from a)
print(c['y'])    # Outputs 2 (from b)
print(c['z'])    # Outputs 3 (from a)
```

## Discussion

A `ChainMap` takes multiple mappings and makes them logically appear as one. However, the mappings are not literally merged together. Instead, a `ChainMap` simply keeps a list of the underlying mappings and redefines common dictionary operations to scan the list. Most operations will work. For example:

```
>>> len(c)
3
>>> list(c.keys())
['x', 'y', 'z']
>>> list(c.values())
[1, 2, 3]
>>>
```

If there are duplicate keys, the values from the first mapping get used. Thus, the entry `c['z']` in the example would always refer to the value in dictionary `a`, not the value in dictionary `b`.

Operations that mutate the mapping always affect the first mapping listed. For example:

```
>>> c['z'] = 10
>>> c['w'] = 40
>>> del c['x']
>>> a
{'w': 40, 'z': 10}
>>> del c['y']
Traceback (most recent call last):
...
KeyError: "Key not found in the first mapping: 'y'"
>>>
```

A `ChainMap` is particularly useful when working with scoped values such as variables in a programming language (i.e., globals, locals, etc.). In fact, there are methods that make this easy:

```

>>> values = ChainMap()
>>> values['x'] = 1
>>> # Add a new mapping
>>> values = values.new_child()
>>> values['x'] = 2
>>> # Add a new mapping
>>> values = values.new_child()
>>> values['x'] = 3
>>> values
ChainMap({'x': 3}, {'x': 2}, {'x': 1})
>>> values['x']
3
>>> # Discard last mapping
>>> values = values.parents
>>> values['x']
2
>>> # Discard last mapping
>>> values = values.parents
>>> values['x']
1
>>> values
ChainMap({'x': 1})
>>>

```

As an alternative to `ChainMap`, you might consider merging dictionaries together using the `update()` method. For example:

```

>>> a = {'x': 1, 'z': 3 }
>>> b = {'y': 2, 'z': 4 }
>>> merged = dict(b)
>>> merged.update(a)
>>> merged['x']
1
>>> merged['y']
2
>>> merged['z']
3
>>>

```

This works, but it requires you to make a completely separate dictionary object (or destructively alter one of the existing dictionaries). Also, if any of the original dictionaries mutate, the changes don't get reflected in the merged dictionary. For example:

```

>>> a['x'] = 13
>>> merged['x']
1

```

A `ChainMap` uses the original dictionaries, so it doesn't have this behavior. For example:

```
>>> a = {'x': 1, 'z': 3 }
>>> b = {'y': 2, 'z': 4 }
>>> merged = ChainMap(a, b)
>>> merged['x']
1
>>> a['x'] = 42
>>> merged['x']  # Notice change to merged dicts
42
>>>
```

---

## CHAPTER 2

# Strings and Text

Almost every useful program involves some kind of text processing, whether it is parsing data or generating output. This chapter focuses on common problems involving text manipulation, such as pulling apart strings, searching, substitution, lexing, and parsing. Many of these tasks can be easily solved using built-in methods of strings. However, more complicated operations might require the use of regular expressions or the creation of a full-fledged parser. All of these topics are covered. In addition, a few tricky aspects of working with Unicode are addressed.

## 2.1. Splitting Strings on Any of Multiple Delimiters

### Problem

You need to split a string into fields, but the delimiters (and spacing around them) aren't consistent throughout the string.

### Solution

The `split()` method of string objects is really meant for very simple cases, and does not allow for multiple delimiters or account for possible whitespace around the delimiters. In cases when you need a bit more flexibility, use the `re.split()` method:

```
>>> line = 'asdf fjdk; afed, fjek,asdf,      foo'
>>> import re
>>> re.split(r'[:,\s]\s*', line)
['asdf', 'fjdk', 'afed', 'fjek', 'asdf', 'foo']
```

### Discussion

The `re.split()` function is useful because you can specify multiple patterns for the separator. For example, as shown in the solution, the separator is either a comma (,),

semicolon (;), or whitespace followed by any amount of extra whitespace. Whenever that pattern is found, the entire match becomes the delimiter between whatever fields lie on either side of the match. The result is a list of fields, just as with `str.split()`.

When using `re.split()`, you need to be a bit careful should the regular expression pattern involve a capture group enclosed in parentheses. If capture groups are used, then the matched text is also included in the result. For example, watch what happens here:

```
>>> fields = re.split(r'(;|\\s)\\s*', line)
>>> fields
['asdf', ' ', 'fjdk', ';', 'afed', ' ', 'fjek', ' ', 'asdf', ' ', 'foo']
>>>
```

Getting the split characters might be useful in certain contexts. For example, maybe you need the split characters later on to reform an output string:

```
>>> values = fields[::2]
>>> delimiters = fields[1::2] + ['']
>>> values
['asdf', 'fjdk', 'afed', 'fjek', 'asdf', 'foo']
>>> delimiters
[' ', ';', ' ', ' ', ' ', ' ', ' ', '']

>>> # Reform the line using the same delimiters
>>> ''.join(v+d for v,d in zip(values, delimiters))
'asdf fjdk;afed,fjek,asdf,foo'
>>>
```

If you don't want the separator characters in the result, but still need to use parentheses to group parts of the regular expression pattern, make sure you use a noncapture group, specified as `(?:...)`. For example:

```
>>> re.split(r'(?:;|\\s)\\s*', line)
['asdf', 'fjdk', 'afed', 'fjek', 'asdf', 'foo']
>>>
```

## 2.2. Matching Text at the Start or End of a String

### Problem

You need to check the start or end of a string for specific text patterns, such as filename extensions, URL schemes, and so on.

### Solution

A simple way to check the beginning or end of a string is to use the `str.starts with()` or `str.endswith()` methods. For example:



```

>>> filename = 'spam.txt'
>>> filename.endswith('.txt')
True
>>> filename.startswith('file:')
False
>>> url = 'http://www.python.org'
>>> url.startswith('http:')
True
>>>

```

If you need to check against multiple choices, simply provide a tuple of possibilities to `startswith()` or `endswith()`:

```

>>> import os
>>> filenames = os.listdir('.')
>>> filenames
[ 'Makefile', 'foo.c', 'bar.py', 'spam.c', 'spam.h' ]
>>> [name for name in filenames if name.endswith(('c', 'h')) ]
['foo.c', 'spam.c', 'spam.h']
>>> any(name.endswith('.py') for name in filenames)
True
>>>

```

Here is another example:

```

from urllib.request import urlopen

def read_data(name):
    if name.startswith(('http:', 'https:', 'ftp:')):
        return urlopen(name).read()
    else:
        with open(name) as f:
            return f.read()

```

Oddly, this is one part of Python where a tuple is actually required as input. If you happen to have the choices specified in a list or set, just make sure you convert them using `tuple()` first. For example:

```

>>> choices = ['http:', 'ftp:']
>>> url = 'http://www.python.org'
>>> url.startswith(choices)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: startswith first arg must be str or a tuple of str, not list
>>> url.startswith(tuple(choices))
True
>>>

```

## Discussion

The `startswith()` and `endswith()` methods provide a very convenient way to perform basic prefix and suffix checking. Similar operations can be performed with slices, but are far less elegant. For example:

```
>>> filename = 'spam.txt'
>>> filename[-4:] == '.txt'
True
>>> url = 'http://www.python.org'
>>> url[:5] == 'http:' or url[:6] == 'https:' or url[:4] == 'ftp:'
True
>>>
```

You might also be inclined to use regular expressions as an alternative. For example:

```
>>> import re
>>> url = 'http://www.python.org'
>>> re.match('http:|https:|ftp:', url)
<_sre.SRE_Match object at 0x101253098>
>>>
```

This works, but is often overkill for simple matching. Using this recipe is simpler and runs faster.

Last, but not least, the `startswith()` and `endswith()` methods look nice when combined with other operations, such as common data reductions. For example, this statement that checks a directory for the presence of certain kinds of files:

```
if any(name.endswith(('.'c', '.h')) for name in listdir(dirname)):
    ...
```

## 2.3. Matching Strings Using Shell Wildcard Patterns

### Problem

You want to match text using the same wildcard patterns as are commonly used when working in Unix shells (e.g., `*.py`, `Dat[0-9]*.csv`, etc.).

### Solution

The `fnmatch` module provides two functions—`fnmatch()` and `fnmatchcase()`—that can be used to perform such matching. The usage is simple:

```
>>> from fnmatch import fnmatch, fnmatchcase
>>> fnmatch('foo.txt', '*.txt')
True
>>> fnmatch('foo.txt', '?oo.txt')
True
>>> fnmatch('Dat45.csv', 'Dat[0-9]*')
```

```
True
>>> names = ['Dat1.csv', 'Dat2.csv', 'config.ini', 'foo.py']
>>> [name for name in names if fnmatch(name, 'Dat*.csv')]
['Dat1.csv', 'Dat2.csv']
>>>
```

Normally, `fnmatch()` matches patterns using the same case-sensitivity rules as the system's underlying filesystem (which varies based on operating system). For example:

```
>>> # On OS X (Mac)
>>> fnmatch('foo.txt', '*.TXT')
False

>>> # On Windows
>>> fnmatch('foo.txt', '*.TXT')
True
>>>
```

If this distinction matters, use `fnmatchcase()` instead. It matches exactly based on the lower- and uppercase conventions that you supply:

```
>>> fnmatchcase('foo.txt', '*.TXT')
False
>>>
```

An often overlooked feature of these functions is their potential use with data processing of nonfilename strings. For example, suppose you have a list of street addresses like this:

```
addresses = [
    '5412 N CLARK ST',
    '1060 W ADDISON ST',
    '1039 W GRANVILLE AVE',
    '2122 N CLARK ST',
    '4802 N BROADWAY',
]
```

You could write list comprehensions like this:

```
>>> from fnmatch import fnmatchcase
>>> [addr for addr in addresses if fnmatchcase(addr, '* ST')]
['5412 N CLARK ST', '1060 W ADDISON ST', '2122 N CLARK ST']
>>> [addr for addr in addresses if fnmatchcase(addr, '54[0-9][0-9] *CLARK*')]
['5412 N CLARK ST']
>>>
```

## Discussion

The matching performed by `fnmatch` sits somewhere between the functionality of simple string methods and the full power of regular expressions. If you're just trying to provide a simple mechanism for allowing wildcards in data processing operations, it's often a reasonable solution.

If you're actually trying to write code that matches filenames, use the `glob` module instead. See [Recipe 5.13](#).

## 2.4. Matching and Searching for Text Patterns

### Problem

You want to match or search text for a specific pattern.

### Solution

If the text you're trying to match is a simple literal, you can often just use the basic string methods, such as `str.find()`, `str.endswith()`, `str.startswith()`, or similar. For example:

```
>>> text = 'yeah, but no, but yeah, but no, but yeah'

>>> # Exact match
>>> text == 'yeah'
False

>>> # Match at start or end
>>> text.startswith('yeah')
True
>>> text.endswith('no')
False

>>> # Search for the location of the first occurrence
>>> text.find('no')
10
>>>
```

For more complicated matching, use regular expressions and the `re` module. To illustrate the basic mechanics of using regular expressions, suppose you want to match dates specified as digits, such as “11/27/2012.” Here is a sample of how you would do it:

```
>>> text1 = '11/27/2012'
>>> text2 = 'Nov 27, 2012'
>>>
>>> import re
>>> # Simple matching: \d+ means match one or more digits
>>> if re.match(r'\d+/\d+/\d+', text1):
...     print('yes')
... else:
...     print('no')
...
yes
>>> if re.match(r'\d+/\d+/\d+', text2):
...     print('yes')
... else:
```

```
...     print('no')
...
no
>>>
```

If you're going to perform a lot of matches using the same pattern, it usually pays to precompile the regular expression pattern into a pattern object first. For example:

```
>>> datepat = re.compile(r'\d+/\d+/\d+')
>>> if datepat.match(text1):
...     print('yes')
... else:
...     print('no')
...
yes
>>> if datepat.match(text2):
...     print('yes')
... else:
...     print('no')
...
no
>>>
```

`match()` always tries to find the match at the start of a string. If you want to search text for all occurrences of a pattern, use the `findall()` method instead. For example:

```
>>> text = 'Today is 11/27/2012. PyCon starts 3/13/2013.'
>>> datepat.findall(text)
['11/27/2012', '3/13/2013']
>>>
```

When defining regular expressions, it is common to introduce capture groups by enclosing parts of the pattern in parentheses. For example:

```
>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)')
>>>
```

Capture groups often simplify subsequent processing of the matched text because the contents of each group can be extracted individually. For example:

```
>>> m = datepat.match('11/27/2012')
>>> m
<_sre.SRE_Match object at 0x1005d2750>

>>> # Extract the contents of each group
>>> m.group(0)
'11/27/2012'
>>> m.group(1)
'11'
>>> m.group(2)
'27'
>>> m.group(3)
'2012'
>>> m.groups()
```

```

('11', '27', '2012')
>>> month, day, year = m.groups()
>>>

>>> # Find all matches (notice splitting into tuples)
>>> text
'Today is 11/27/2012. PyCon starts 3/13/2013.'
>>> datepat.findall(text)
[('11', '27', '2012'), ('3', '13', '2013')]
>>> for month, day, year in datepat.findall(text):
...     print('{}-{}-{}'.format(year, month, day))
...
2012-11-27
2013-3-13
>>>

```

The `findall()` method searches the text and finds all matches, returning them as a list. If you want to find matches iteratively, use the `finditer()` method instead. For example:

```

>>> for m in datepat.finditer(text):
...     print(m.groups())
...
('11', '27', '2012')
('3', '13', '2013')
>>>

```

## Discussion

A basic tutorial on the theory of regular expressions is beyond the scope of this book. However, this recipe illustrates the absolute basics of using the `re` module to match and search for text. The essential functionality is first compiling a pattern using `re.compile()` and then using methods such as `match()`, `findall()`, or `finditer()`.

When specifying patterns, it is relatively common to use raw strings such as `r'(\d+)/(\d+)/(\d+)'`. Such strings leave the backslash character uninterpreted, which can be useful in the context of regular expressions. Otherwise, you need to use double backslashes such as `'(\\d+)/ (\\d+)/ (\\d+)'`.

Be aware that the `match()` method only checks the beginning of a string. It's possible that it will match things you aren't expecting. For example:

```

>>> m = datepat.match('11/27/2012abcdef')
>>> m
<_sre.SRE_Match object at 0x1005d27e8>
>>> m.group()
'11/27/2012'
>>>

```

If you want an exact match, make sure the pattern includes the end-marker (`$`), as in the following:

```
>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)$')
>>> datepat.match('11/27/2012abcdef')
>>> datepat.match('11/27/2012')
<_sre.SRE_Match object at 0x1005d2750>
>>>
```

Last, if you're just doing a simple text matching/searching operation, you can often skip the compilation step and use module-level functions in the `re` module instead. For example:

```
>>> re.findall(r'(\d+)/(\d+)/(\d+)', text)
[('11', '27', '2012'), ('3', '13', '2013')]
>>>
```

Be aware, though, that if you're going to perform a lot of matching or searching, it usually pays to compile the pattern first and use it over and over again. The module-level functions keep a cache of recently compiled patterns, so there isn't a huge performance hit, but you'll save a few lookups and extra processing by using your own compiled pattern.

## 2.5. Searching and Replacing Text

### Problem

You want to search for and replace a text pattern in a string.

### Solution

For simple literal patterns, use the `str.replace()` method. For example:

```
>>> text = 'yeah, but no, but yeah, but no, but yeah'
>>> text.replace('yeah', 'yep')
'yep, but no, but yep, but no, but yep'
>>>
```

For more complicated patterns, use the `sub()` functions/methods in the `re` module. To illustrate, suppose you want to rewrite dates of the form “11/27/2012” as “2012-11-27.” Here is a sample of how to do it:

```
>>> text = 'Today is 11/27/2012. PyCon starts 3/13/2013.'
>>> import re
>>> re.sub(r'(\d+)/(\d+)/(\d+)', r'\3-\1-\2', text)
'Today is 2012-11-27. PyCon starts 2013-3-13.'
>>>
```

The first argument to `sub()` is the pattern to match and the second argument is the replacement pattern. Backslashed digits such as `\3` refer to capture group numbers in the pattern.

If you're going to perform repeated substitutions of the same pattern, consider compiling it first for better performance. For example:

```
>>> import re
>>> datepat = re.compile(r'(\d+)/(\d+)/(\d+)')
>>> datepat.sub(r'\3-\1-\2', text)
'Today is 2012-11-27. PyCon starts 2013-3-13.'
>>>
```

For more complicated substitutions, it's possible to specify a substitution callback function instead. For example:

```
>>> from calendar import month_abbr
>>> def change_date(m):
...     mon_name = month_abbr[int(m.group(1))]
...     return '{} {} {}'.format(m.group(2), mon_name, m.group(3))
...
>>> datepat.sub(change_date, text)
'Today is 27 Nov 2012. PyCon starts 13 Mar 2013.'
>>>
```

As input, the argument to the substitution callback is a match object, as returned by `match()` or `find()`. Use the `.group()` method to extract specific parts of the match. The function should return the replacement text.

If you want to know how many substitutions were made in addition to getting the replacement text, use `re.subn()` instead. For example:

```
>>> newtext, n = datepat.subn(r'\3-\1-\2', text)
>>> newtext
'Today is 2012-11-27. PyCon starts 2013-3-13.'
>>> n
2
>>>
```

## Discussion

There isn't much more to regular expression search and replace than the `sub()` method shown. The trickiest part is specifying the regular expression pattern—something that's best left as an exercise to the reader.

## 2.6. Searching and Replacing Case-Insensitive Text

### Problem

You need to search for and possibly replace text in a case-insensitive manner.



## Solution

To perform case-insensitive text operations, you need to use the `re` module and supply the `re.IGNORECASE` flag to various operations. For example:

```
>>> text = 'UPPER PYTHON, lower python, Mixed Python'
>>> re.findall('python', text, flags=re.IGNORECASE)
['PYTHON', 'python', 'Python']
>>> re.sub('python', 'snake', text, flags=re.IGNORECASE)
'UPPER snake, lower snake, Mixed snake'
>>>
```

The last example reveals a limitation that replacing text won't match the case of the matched text. If you need to fix this, you might have to use a support function, as in the following:

```
def matchcase(word):
    def replace(m):
        text = m.group()
        if text.isupper():
            return word.upper()
        elif text.islower():
            return word.lower()
        elif text[0].isupper():
            return word.capitalize()
        else:
            return word
    return replace
```

Here is an example of using this last function:

```
>>> re.sub('python', matchcase('snake'), text, flags=re.IGNORECASE)
'UPPER SNAKE, lower snake, Mixed Snake'
>>>
```

## Discussion

For simple cases, simply providing the `re.IGNORECASE` is enough to perform case-insensitive matching. However, be aware that this may not be enough for certain kinds of Unicode matching involving case folding. See [Recipe 2.10](#) for more details.

## 2.7. Specifying a Regular Expression for the Shortest Match

### Problem

You're trying to match a text pattern using regular expressions, but it is identifying the longest possible matches of a pattern. Instead, you would like to change it to find the shortest possible match.

## Solution

This problem often arises in patterns that try to match text enclosed inside a pair of starting and ending delimiters (e.g., a quoted string). To illustrate, consider this example:

```
>>> str_pat = re.compile(r'\"(.*)\"')
>>> text1 = 'Computer says "no."'
>>> str_pat.findall(text1)
['no.']
>>> text2 = 'Computer says "no." Phone says "yes."'
>>> str_pat.findall(text2)
['no." Phone says "yes.']
>>>
```

In this example, the pattern `r'\"(.*)\"'` is attempting to match text enclosed inside quotes. However, the `*` operator in a regular expression is greedy, so matching is based on finding the longest possible match. Thus, in the second example involving `text2`, it incorrectly matches the two quoted strings.

To fix this, add the `?` modifier after the `*` operator in the pattern, like this:

```
>>> str_pat = re.compile(r'\"(.*)?\"')
>>> str_pat.findall(text2)
['no.', 'yes.']
>>>
```

This makes the matching nongreedy, and produces the shortest match instead.

## Discussion

This recipe addresses one of the more common problems encountered when writing regular expressions involving the dot (`.`) character. In a pattern, the dot matches any character except a newline. However, if you bracket the dot with starting and ending text (such as a quote), matching will try to find the longest possible match to the pattern. This causes multiple occurrences of the starting or ending text to be skipped altogether and included in the results of the longer match. Adding the `?` right after operators such as `*` or `+` forces the matching algorithm to look for the shortest possible match instead.

## 2.8. Writing a Regular Expression for Multiline Patterns

### Problem

You're trying to match a block of text using a regular expression, but you need the match to span multiple lines.

## Solution

This problem typically arises in patterns that use the dot (.) to match any character but forget to account for the fact that it doesn't match newlines. For example, suppose you are trying to match C-style comments:

```
>>> comment = re.compile(r'/*(.*?)*/')
>>> text1 = '/* this is a comment */'
>>> text2 = '''/* this is a
...           multiline comment */
... '''
>>>
>>> comment.findall(text1)
[' this is a comment ']
>>> comment.findall(text2)
[]
>>>
```

To fix the problem, you can add support for newlines. For example:

```
>>> comment = re.compile(r'/*((?:.|\\n)*)*/')
>>> comment.findall(text2)
[' this is a\\n           multiline comment ']
>>>
```

In this pattern, (?:.|\\n) specifies a noncapture group (i.e., it defines a group for the purposes of matching, but that group is not captured separately or numbered).

## Discussion

The `re.compile()` function accepts a flag, `re.DOTALL`, which is useful here. It makes the `.` in a regular expression match all characters, including newlines. For example:

```
>>> comment = re.compile(r'/*(.*?)*/', re.DOTALL)
>>> comment.findall(text2)
[' this is a\\n           multiline comment ']
```

Using the `re.DOTALL` flag works fine for simple cases, but might be problematic if you're working with extremely complicated patterns or a mix of separate regular expressions that have been combined together for the purpose of tokenizing, as described in [Recipe 2.18](#). If given a choice, it's usually better to define your regular expression pattern so that it works correctly without the need for extra flags.

## 2.9. Normalizing Unicode Text to a Standard Representation

### Problem

You’re working with Unicode strings, but need to make sure that all of the strings have the same underlying representation.

### Solution

In Unicode, certain characters can be represented by more than one valid sequence of code points. To illustrate, consider the following example:

```
>>> s1 = 'Spicy Jalape\u00f1o'
>>> s2 = 'Spicy Jalapen\u0303o'
>>> s1
'Spicy Jalapeño'
>>> s2
'Spicy Jalapeño'
>>> s1 == s2
False
>>> len(s1)
14
>>> len(s2)
15
>>>
```

Here the text “Spicy Jalapeño” has been presented in two forms. The first uses the fully composed “ñ” character (U+00F1). The second uses the Latin letter “n” followed by a “~” combining character (U+0303).

Having multiple representations is a problem for programs that compare strings. In order to fix this, you should first normalize the text into a standard representation using the `unicodedata` module:

```
>>> import unicodedata
>>> t1 = unicodedata.normalize('NFC', s1)
>>> t2 = unicodedata.normalize('NFC', s2)
>>> t1 == t2
True
>>> print(ascii(t1))
'Spicy Jalape\x1f1o'

>>> t3 = unicodedata.normalize('NFD', s1)
>>> t4 = unicodedata.normalize('NFD', s2)
>>> t3 == t4
True
>>> print(ascii(t3))
'Spicy Jalapen\u0303o'
>>>
```

The first argument to `normalize()` specifies how you want the string normalized. NFC means that characters should be fully composed (i.e., use a single code point if possible). NFD means that characters should be fully decomposed with the use of combining characters.

Python also supports the normalization forms NFKC and NFKD, which add extra compatibility features for dealing with certain kinds of characters. For example:

```
>>> s = '\ufb01' # A single character
>>> s
'fi'
>>> unicodedata.normalize('NFD', s)
'fi'

# Notice how the combined letters are broken apart here
>>> unicodedata.normalize('NFKD', s)
'fi'
>>> unicodedata.normalize('NFKC', s)
'fi'
>>>
```

## Discussion

Normalization is an important part of any code that needs to ensure that it processes Unicode text in a sane and consistent way. This is especially true when processing strings received as part of user input where you have little control of the encoding.

Normalization can also be an important part of sanitizing and filtering text. For example, suppose you want to remove all diacritical marks from some text (possibly for the purposes of searching or matching):

```
>>> t1 = unicodedata.normalize('NFD', s1)
>>> ''.join(c for c in t1 if not unicodedata.combining(c))
'Spicy Jalapeno'
>>>
```

This last example shows another important aspect of the `unicodedata` module—namely, utility functions for testing characters against character classes. The `combining()` function tests a character to see if it is a combining character. There are other functions in the module for finding character categories, testing digits, and so forth.

Unicode is obviously a large topic. For more detailed reference information about normalization, visit [Unicode's page on the subject](#). Ned Batchelder has also given an excellent presentation on Python Unicode handling issues at [his website](#).

## 2.10. Working with Unicode Characters in Regular Expressions

### Problem

You are using regular expressions to process text, but are concerned about the handling of Unicode characters.

### Solution

By default, the `re` module is already programmed with rudimentary knowledge of certain Unicode character classes. For example, `\d` already matches any unicode digit character:

```
>>> import re
>>> num = re.compile('\d+')
>>> # ASCII digits
>>> num.match('123')
<_sre.SRE_Match object at 0x1007d9ed0>

>>> # Arabic digits
>>> num.match('\u0661\u0662\u0663')
<_sre.SRE_Match object at 0x101234030>
>>>
```

If you need to include specific Unicode characters in patterns, you can use the usual escape sequence for Unicode characters (e.g., `\uFFFF` or `\UFFFFFF`). For example, here is a regex that matches all characters in a few different Arabic code pages:

```
>>> arabic = re.compile('[\u0600-\u06ff\u0750-\u077f\u08a0-\u08ff]+')
>>>
```

When performing matching and searching operations, it's a good idea to normalize and possibly sanitize all text to a standard form first (see [Recipe 2.9](#)). However, it's also important to be aware of special cases. For example, consider the behavior of case-insensitive matching combined with case folding:

```
>>> pat = re.compile('stra\u00dfe', re.IGNORECASE)
>>> s = 'straße'
>>> pat.match(s) # Matches
<_sre.SRE_Match object at 0x10069d370>
>>> pat.match(s.upper()) # Doesn't match
>>> s.upper() # Case folds
'STRASSE'
>>>
```

## Discussion

Mixing Unicode and regular expressions is often a good way to make your head explode. If you're going to do it seriously, you should consider installing the third-party **regex library**, which provides full support for Unicode case folding, as well as a variety of other interesting features, including approximate matching.

## 2.11. Stripping Unwanted Characters from Strings

### Problem

You want to strip unwanted characters, such as whitespace, from the beginning, end, or middle of a text string.

### Solution

The `strip()` method can be used to strip characters from the beginning or end of a string. `lstrip()` and `rstrip()` perform stripping from the left or right side, respectively. By default, these methods strip whitespace, but other characters can be given. For example:

```
>>> # Whitespace stripping
>>> s = '  hello world \n'
>>> s.strip()
'hello world'
>>> s.lstrip()
'hello world \n'
>>> s.rstrip()
'  hello world'
>>>

>>> # Character stripping
>>> t = '-----hello===='
>>> t.lstrip('-')
'hello===='
>>> t.strip('-=')
'hello'
>>>
```

## Discussion

The various `strip()` methods are commonly used when reading and cleaning up data for later processing. For example, you can use them to get rid of whitespace, remove quotations, and other tasks.

Be aware that stripping does not apply to any text in the middle of a string. For example:

```

>>> s = ' hello      world \n'
>>> s = s.strip()
>>> s
'hello      world'
>>>

```

If you needed to do something to the inner space, you would need to use another technique, such as using the `replace()` method or a regular expression substitution. For example:

```

>>> s.replace(' ', '')
'helloworld'
>>> import re
>>> re.sub('\s+', ' ', s)
'hello world'
>>>

```

It is often the case that you want to combine string stripping operations with some other kind of iterative processing, such as reading lines of data from a file. If so, this is one area where a generator expression can be useful. For example:

```

with open(filename) as f:
    lines = (line.strip() for line in f)
    for line in lines:
        ...

```

Here, the expression `lines = (line.strip() for line in f)` acts as a kind of data transform. It's efficient because it doesn't actually read the data into any kind of temporary list first. It just creates an iterator where all of the lines produced have the stripping operation applied to them.

For even more advanced stripping, you might turn to the `translate()` method. See the next recipe on sanitizing strings for further details.

## 2.12. Sanitizing and Cleaning Up Text

### Problem

Some bored script kiddie has entered the text “pýthöñ” into a form on your web page and you'd like to clean it up somehow.

### Solution

The problem of sanitizing and cleaning up text applies to a wide variety of problems involving text parsing and data handling. At a very simple level, you might use basic string functions (e.g., `str.upper()` and `str.lower()`) to convert text to a standard case. Simple replacements using `str.replace()` or `re.sub()` can focus on removing or



changing very specific character sequences. You can also normalize text using `unicode.data.normalize()`, as shown in [Recipe 2.9](#).

However, you might want to take the sanitation process a step further. Perhaps, for example, you want to eliminate whole ranges of characters or strip diacritical marks. To do so, you can turn to the often overlooked `str.translate()` method. To illustrate, suppose you've got a messy string such as the following:

```
>>> s = 'pýthöñ\fis\tawesome\r\n'
>>> s
'pýthöñ\x0cis\tawesome\r\n'
>>>
```

The first step is to clean up the whitespace. To do this, make a small translation table and use `translate()`:

```
>>> remap = {
...     ord('\t') : ' ',
...     ord('\f') : ' ',
...     ord('\r') : None      # Deleted
... }
>>> a = s.translate(remap)
>>> a
'pýthöñ is awesome\n'
>>>
```

As you can see here, whitespace characters such as `\t` and `\f` have been remapped to a single space. The carriage return `\r` has been deleted entirely.

You can take this remapping idea a step further and build much bigger tables. For example, let's remove all combining characters:

```
>>> import unicodedata
>>> import sys
>>> cmb_chrs = dict.fromkeys(c for c in range(sys.maxunicode)
...                          if unicodedata.combining(chr(c)))
...
>>> b = unicodedata.normalize('NFD', a)
>>> b
'pýthöñ is awesome\n'
>>> b.translate(cmb_chrs)
'python is awesome\n'
>>>
```

In this last example, a dictionary mapping every Unicode combining character to `None` is created using the `dict.fromkeys()`.

The original input is then normalized into a decomposed form using `unicodedata.normalize()`. From there, the `translate` function is used to delete all of the accents. Similar techniques can be used to remove other kinds of characters (e.g., control characters, etc.).

As another example, here is a translation table that maps all Unicode decimal digit characters to their equivalent in ASCII:

```
>>> digitmap = { c: ord('0') + unicodedata.digit(chr(c))
...               for c in range(sys.maxunicode)
...               if unicodedata.category(chr(c)) == 'Nd' }
...
>>> len(digitmap)
460
>>> # Arabic digits
>>> x = '\u0661\u0662\u0663'
>>> x.translate(digitmap)
'123'
>>>
```

Yet another technique for cleaning up text involves I/O decoding and encoding functions. The idea here is to first do some preliminary cleanup of the text, and then run it through a combination of `encode()` or `decode()` operations to strip or alter it. For example:

```
>>> a
'pýthõñ is awesome\n'
>>> b = unicodedata.normalize('NFD', a)
>>> b.encode('ascii', 'ignore').decode('ascii')
'python is awesome\n'
>>>
```

Here the normalization process decomposed the original text into characters along with separate combining characters. The subsequent ASCII encoding/decoding simply discarded all of those characters in one fell swoop. Naturally, this would only work if getting an ASCII representation was the final goal.

## Discussion

A major issue with sanitizing text can be runtime performance. As a general rule, the simpler it is, the faster it will run. For simple replacements, the `str.replace()` method is often the fastest approach—even if you have to call it multiple times. For instance, to clean up whitespace, you could use code like this:

```
def clean_spaces(s):
    s = s.replace('\r', ' ')
    s = s.replace('\t', ' ')
    s = s.replace('\f', ' ')
    return s
```

If you try it, you'll find that it's quite a bit faster than using `translate()` or an approach using a regular expression.

On the other hand, the `translate()` method is very fast if you need to perform any kind of nontrivial character-to-character remapping or deletion.

In the big picture, performance is something you will have to study further in your particular application. Unfortunately, it's impossible to suggest one specific technique that works best for all cases, so try different approaches and measure it.

Although the focus of this recipe has been text, similar techniques can be applied to bytes, including simple replacements, translation, and regular expressions.

## 2.13. Aligning Text Strings

### Problem

You need to format text with some sort of alignment applied.

### Solution

For basic alignment of strings, the `ljust()`, `rjust()`, and `center()` methods of strings can be used. For example:

```
>>> text = 'Hello World'
>>> text.ljust(20)
'Hello World      '
>>> text.rjust(20)
'      Hello World'
>>> text.center(20)
'    Hello World   '
>>>
```

All of these methods accept an optional `&&65.180&&fill` character as well. For example:

```
>>> text.rjust(20, '=')
'=====Hello World'
>>> text.center(20, '*')
'****Hello World****'
>>>
```

The `format()` function can also be used to easily align things. All you need to do is use the `<`, `>`, or `^` characters along with a desired width. For example:

```
>>> format(text, '>20')
'      Hello World'
>>> format(text, '<20')
'Hello World      '
>>> format(text, '^20')
'    Hello World   '
>>>
```

If you want to include a fill character other than a space, specify it before the alignment character:

```
>>> format(text, '=>20s')
'=====Hello World'
```

```
>>> format(text, '*^20s')
'****Hello World*****'
>>>
```

These format codes can also be used in the `format()` method when formatting multiple values. For example:

```
>>> '{:>10s} {:>10s}'.format('Hello', 'World')
'      Hello      World'
>>>
```

One benefit of `format()` is that it is not specific to strings. It works with any value, making it more general purpose. For instance, you can use it with numbers:

```
>>> x = 1.2345
>>> format(x, '>10')
'      1.2345'
>>> format(x, '^10.2f')
'      1.23      '
>>>
```

## Discussion

In older code, you will also see the `%` operator used to format text. For example:

```
>>> '%-20s' % text
'Hello World      '
>>> '%20s' % text
'      Hello World'
>>>
```

However, in new code, you should probably prefer the use of the `format()` function or method. `format()` is a lot more powerful than what is provided with the `%` operator. Moreover, `format()` is more general purpose than using the `ljust()`, `rjust()`, or `center()` method of strings in that it works with any kind of object.

For a complete list of features available with the `format()` function, consult [the online Python documentation](#).

## 2.14. Combining and Concatenating Strings

### Problem

You want to combine many small strings together into a larger string.

### Solution

If the strings you wish to combine are found in a sequence or iterable, the fastest way to combine them is to use the `join()` method. For example:

```

>>> parts = ['Is', 'Chicago', 'Not', 'Chicago?']
>>> ' '.join(parts)
'Is Chicago Not Chicago?'
>>> ','.join(parts)
'Is,Chicago,Not,Chicago?'
>>> ''.join(parts)
'IsChicagoNotChicago?'
>>>

```

At first glance, this syntax might look really odd, but the `join()` operation is specified as a method on strings. Partly this is because the objects you want to join could come from any number of different data sequences (e.g., lists, tuples, dicts, files, sets, or generators), and it would be redundant to have `join()` implemented as a method on all of those objects separately. So you just specify the separator string that you want and use the `join()` method on it to glue text fragments together.

If you're only combining a few strings, using `+` usually works well enough:

```

>>> a = 'Is Chicago'
>>> b = 'Not Chicago?'
>>> a + ' ' + b
'Is Chicago Not Chicago?'
>>>

```

The `+` operator also works fine as a substitute for more complicated string formatting operations. For example:

```

>>> print('{} {}'.format(a,b))
Is Chicago Not Chicago?
>>> print(a + ' ' + b)
Is Chicago Not Chicago?
>>>

```

If you're trying to combine string literals together in source code, you can simply place them adjacent to each other with no `+` operator. For example:

```

>>> a = 'Hello' 'World'
>>> a
'HelloWorld'
>>>

```

## Discussion

Joining strings together might not seem advanced enough to warrant an entire recipe, but it's often an area where programmers make programming choices that severely impact the performance of their code.

The most important thing to know is that using the `+` operator to join a lot of strings together is grossly inefficient due to the memory copies and garbage collection that occurs. In particular, you never want to write code that joins strings together like this:

```
s = ''
for p in parts:
    s += p
```

This runs quite a bit slower than using the `join()` method, mainly because each `+=` operation creates a new string object. You're better off just collecting all of the parts first and then joining them together at the end.

One related (and pretty neat) trick is the conversion of data to strings and concatenation at the same time using a generator expression, as described in [Recipe 1.19](#). For example:

```
>>> data = ['ACME', 50, 91.1]
>>> ','.join(str(d) for d in data)
'ACME,50,91.1'
>>>
```

Also be on the lookout for unnecessary string concatenations. Sometimes programmers get carried away with concatenation when it's really not technically necessary. For example, when printing:

```
print(a + ':' + b + ':' + c)      # Ugly
print(':%.join([a, b, c]))      # Still ugly

print(a, b, c, sep=':')         # Better
```

Mixing I/O operations and string concatenation is something that might require study in your application. For example, consider the following two code fragments:

```
# Version 1 (string concatenation)
f.write(chunk1 + chunk2)

# Version 2 (separate I/O operations)
f.write(chunk1)
f.write(chunk2)
```

If the two strings are small, the first version might offer much better performance due to the inherent expense of carrying out an I/O system call. On the other hand, if the two strings are large, the second version may be more efficient, since it avoids making a large temporary result and copying large blocks of memory around. Again, it must be stressed that this is something you would have to study in relation to your own data in order to determine which performs best.

Last, but not least, if you're writing code that is building output from lots of small strings, you might consider writing that code as a generator function, using `yield` to emit fragments. For example:

```
def sample():
    yield 'Is'
    yield 'Chicago'
    yield 'Not'
    yield 'Chicago?'
```

The interesting thing about this approach is that it makes no assumption about how the fragments are to be assembled together. For example, you could simply join the fragments using `join()`:

```
text = ''.join(sample())
```

Or you could redirect the fragments to I/O:

```
for part in sample():
    f.write(part)
```

Or you could come up with some kind of hybrid scheme that's smart about combining I/O operations:

```
def combine(source, maxsize):
    parts = []
    size = 0
    for part in source:
        parts.append(part)
        size += len(part)
        if size > maxsize:
            yield ''.join(parts)
            parts = []
            size = 0
    yield ''.join(parts)

for part in combine(sample(), 32768):
    f.write(part)
```

The key point is that the original generator function doesn't have to know the precise details. It just yields the parts.

## 2.15. Interpolating Variables in Strings

### Problem

You want to create a string in which embedded variable names are substituted with a string representation of a variable's value.

### Solution

Python has no direct support for simply substituting variable values in strings. However, this feature can be approximated using the `format()` method of strings. For example:

```
>>> s = '{name} has {n} messages.'
>>> s.format(name='Guido', n=37)
'Guido has 37 messages.'
>>>
```

Alternatively, if the values to be substituted are truly found in variables, you can use the combination of `format_map()` and `vars()`, as in the following:

```
>>> name = 'Guido'
>>> n = 37
>>> s.format_map(vars())
'Guido has 37 messages.'
>>>
```

One subtle feature of `vars()` is that it also works with instances. For example:

```
>>> class Info:
...     def __init__(self, name, n):
...         self.name = name
...         self.n = n
...
>>> a = Info('Guido', 37)
>>> s.format_map(vars(a))
'Guido has 37 messages.'
>>>
```

One downside of `format()` and `format_map()` is that they do not deal gracefully with missing values. For example:

```
>>> s.format(name='Guido')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'n'
>>>
```

One way to avoid this is to define an alternative dictionary class with a `__missing__()` method, as in the following:

```
class safesub(dict):
    def __missing__(self, key):
        return '{' + key + '}'
```

Now use this class to wrap the inputs to `format_map()`:

```
>>> del n      # Make sure n is undefined
>>> s.format_map(safesub(vars()))
'Guido has {n} messages.'
>>>
```

If you find yourself frequently performing these steps in your code, you could hide the variable substitution process behind a small utility function that employs a so-called “frame hack.” For example:

```
import sys

def sub(text):
    return text.format_map(safesub(sys._getframe(1).f_locals))
```

Now you can type things like this:



```

>>> name = 'Guido'
>>> n = 37
>>> print(sub('Hello {name}'))
Hello Guido
>>> print(sub('You have {n} messages.'))
You have 37 messages.
>>> print(sub('Your favorite color is {color}'))
Your favorite color is {color}
>>>

```

## Discussion

The lack of true variable interpolation in Python has led to a variety of solutions over the years. As an alternative to the solution presented in this recipe, you will sometimes see string formatting like this:

```

>>> name = 'Guido'
>>> n = 37
>>> '%(name) has %(n) messages.' % vars()
'Guido has 37 messages.'
>>>

```

You may also see the use of template strings:

```

>>> import string
>>> s = string.Template('$name has $n messages.')
>>> s.substitute(vars())
'Guido has 37 messages.'
>>>

```

However, the `format()` and `format_map()` methods are more modern than either of these alternatives, and should be preferred. One benefit of using `format()` is that you also get all of the features related to string formatting (alignment, padding, numerical formatting, etc.), which is simply not possible with alternatives such as `Template` string objects.

Parts of this recipe also illustrate a few interesting advanced features. The little-known `__missing__()` method of mapping/dict classes is a method that you can define to handle missing values. In the `safesub` class, this method has been defined to return missing values back as a placeholder. Instead of getting a `KeyError` exception, you would see the missing values appearing in the resulting string (potentially useful for debugging).

The `sub()` function uses `sys._getframe(1)` to return the stack frame of the caller. From that, the `f_locals` attribute is accessed to get the local variables. It goes without saying that messing around with stack frames should probably be avoided in most code. However, for utility functions such as a string substitution feature, it can be useful. As an aside, it's probably worth noting that `f_locals` is a dictionary that is a copy of the local variables in the calling function. Although you can modify the contents of `f_locals`,

the modifications don't actually have any lasting effect. Thus, even though accessing a different stack frame might look evil, it's not possible to accidentally overwrite variables or change the local environment of the caller.

## 2.16. Reformatting Text to a Fixed Number of Columns

### Problem

You have long strings that you want to reformat so that they fill a user-specified number of columns.

### Solution

Use the `textwrap` module to reformat text for output. For example, suppose you have the following long string:

```
s = "Look into my eyes, look into my eyes, the eyes, the eyes, \
the eyes, not around the eyes, don't look around the eyes, \
look into my eyes, you're under."
```

Here's how you can use the `textwrap` module to reformat it in various ways:

```
>>> import textwrap
>>> print(textwrap.fill(s, 70))
Look into my eyes, look into my eyes, the eyes, the eyes, the eyes,
not around the eyes, don't look around the eyes, look into my eyes,
you're under.
```

```
>>> print(textwrap.fill(s, 40))
Look into my eyes, look into my eyes,
the eyes, the eyes, the eyes, not around
the eyes, don't look around the eyes,
look into my eyes, you're under.
```

```
>>> print(textwrap.fill(s, 40, initial_indent='    '))
    Look into my eyes, look into my
eyes, the eyes, the eyes, the eyes, not
around the eyes, don't look around the
eyes, look into my eyes, you're under.
```

```
>>> print(textwrap.fill(s, 40, subsequent_indent='    '))
Look into my eyes, look into my eyes,
    the eyes, the eyes, the eyes, not
    around the eyes, don't look around
    the eyes, look into my eyes, you're
    under.
```

## Discussion

The `textwrap` module is a straightforward way to clean up text for printing—especially if you want the output to fit nicely on the terminal. On the subject of the terminal size, you can obtain it using `os.get_terminal_size()`. For example:

```
>>> import os
>>> os.get_terminal_size().columns
80
>>>
```

The `fill()` method has a few additional options that control how it handles tabs, sentence endings, and so on. Look at the [documentation for the `textwrap.TextWrapper` class](#) for further details.

## 2.17. Handling HTML and XML Entities in Text

### Problem

You want to replace HTML or XML entities such as `&entity;` or `&#code;` with their corresponding text. Alternatively, you need to produce text, but escape certain characters (e.g., `<`, `>`, or `&`).

### Solution

If you are producing text, replacing special characters such as `<` or `>` is relatively easy if you use the `html.escape()` function. For example:

```
>>> s = 'Elements are written as "<tag>text</tag>". '
>>> import html
>>> print(s)
Elements are written as "<tag>text</tag>".
>>> print(html.escape(s))
Elements are written as "&lt;tag&gt;text&lt;/tag&gt;&quot;".

>>> # Disable escaping of quotes
>>> print(html.escape(s, quote=False))
Elements are written as "&lt;tag&gt;text&lt;/tag&gt;".
>>>
```

If you're trying to emit text as ASCII and want to embed character code entities for non-ASCII characters, you can use the `errors='xmlcharrefreplace'` argument to various I/O-related functions to do it. For example:

```
>>> s = 'Spicy Jalapeño'
>>> s.encode('ascii', errors='xmlcharrefreplace')
b'Spicy Jalape&#241;o'
>>>
```

To replace entities in text, a different approach is needed. If you're actually processing HTML or XML, try using a proper HTML or XML parser first. Normally, these tools will automatically take care of replacing the values for you during parsing and you don't need to worry about it.

If, for some reason, you've received bare text with some entities in it and you want them replaced manually, you can usually do it using various utility functions/methods associated with HTML or XML parsers. For example:

```
>>> s = 'Spicy &quot;Jalape&#241;o&quot;.'
>>> from html.parser import HTMLParser
>>> p = HTMLParser()
>>> p.unescape(s)
'Spicy "Jalapeño".'
>>>

>>> t = 'The prompt is &gt;&gt;&gt;'
>>> from xml.sax.saxutils import unescape
>>> unescape(t)
'The prompt is >>>'
>>>
```

## Discussion

Proper escaping of special characters is an easily overlooked detail of generating HTML or XML. This is especially true if you're generating such output yourself using `print()` or other basic string formatting features. Using a utility function such as `html.escape()` is an easy solution.

If you need to process text in the other direction, various utility functions, such as `xml.sax.saxutils.unescape()`, can help. However, you really need to investigate the use of a proper parser. For example, if processing HTML or XML, using a parsing module such as `html.parser` or `xml.etree.ElementTree` should already take care of details related to replacing entities in the input text for you.

## 2.18. Tokenizing Text

### Problem

You have a string that you want to parse left to right into a stream of tokens.

### Solution

Suppose you have a string of text such as this:

```
text = 'foo = 23 + 42 * 10'
```

To tokenize the string, you need to do more than merely match patterns. You need to have some way to identify the kind of pattern as well. For instance, you might want to turn the string into a sequence of pairs like this:

```
tokens = [('NAME', 'foo'), ('EQ', '='), ('NUM', '23'), ('PLUS', '+'),
          ('NUM', '42'), ('TIMES', '*'), ('NUM', '10')]
```

To do this kind of splitting, the first step is to define all of the possible tokens, including whitespace, by regular expression patterns using named capture groups such as this:

```
import re
NAME = r'(?P<NAME>[a-zA-Z_][a-zA-Z_0-9]*)'
NUM = r'(?P<NUM>\d+)'
PLUS = r'(?P<PLUS>\+)'
TIMES = r'(?P<TIMES>\*)'
EQ = r'(?P<EQ>=)'
WS = r'(?P<WS>\s+)'

master_pat = re.compile('|'.join([NAME, NUM, PLUS, TIMES, EQ, WS]))
```

In these re patterns, the `?P<TOKENNAME>` convention is used to assign a name to the pattern. This will be used later.

Next, to tokenize, use the little-known `scanner()` method of pattern objects. This method creates a scanner object in which repeated calls to `match()` step through the supplied text one match at a time. Here is an interactive example of how a scanner object works:

```
>>> scanner = master_pat.scanner('foo = 42')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('NAME', 'foo')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('WS', ' ')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('EQ', '=')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('WS', ' ')
>>> scanner.match()
<_sre.SRE_Match object at 0x100677738>
>>> _.lastgroup, _.group()
('NUM', '42')
>>> scanner.match()
>>>
```

To take this technique and put it into code, it can be cleaned up and easily packaged into a generator like this:

```
from collections import namedtuple

Token = namedtuple('Token', ['type', 'value'])

def generate_tokens(pat, text):
    scanner = pat.scanner(text)
    for m in iter(scanner.match, None):
        yield Token(m.lastgroup, m.group())

# Example use
for tok in generate_tokens(master_pat, 'foo = 42'):
    print(tok)

# Produces output
# Token(type='NAME', value='foo')
# Token(type='WS', value=' ')
# Token(type='EQ', value='=')
# Token(type='WS', value=' ')
# Token(type='NUM', value='42')
```

If you want to filter the token stream in some way, you can either define more generator functions or use a generator expression. For example, here is how you might filter out all whitespace tokens.

```
tokens = (tok for tok in generate_tokens(master_pat, text)
           if tok.type != 'WS')
for tok in tokens:
    print(tok)
```

## Discussion

Tokenizing is often the first step for more advanced kinds of text parsing and handling. To use the scanning technique shown, there are a few important details to keep in mind. First, you must make sure that you identify every possible text sequence that might appear in the input with a corresponding re pattern. If any nonmatching text is found, scanning simply stops. This is why it was necessary to specify the whitespace (WS) token in the example.

The order of tokens in the master regular expression also matters. When matching, re tries to match patterns in the order specified. Thus, if a pattern happens to be a substring of a longer pattern, you need to make sure the longer pattern goes first. For example:

```
LT = r'(?P<LT><)'
LE = r'(?P<LE><=)'
EQ = r'(?P<EQ>=)'

master_pat = re.compile('|'.join([LE, LT, EQ])) # Correct
# master_pat = re.compile('|'.join([LT, LE, EQ])) # Incorrect
```

The second pattern is wrong because it would match the text `<=` as the token `LT` followed by the token `EQ`, not the single token `LE`, as was probably desired.

Last, but not least, you need to watch out for patterns that form substrings. For example, suppose you have two patterns like this:

```
PRINT = r'(P<PRINT>print)'  
NAME  = r'(P<NAME>[a-zA-Z_][a-zA-Z_0-9]*)'  
  
master_pat = re.compile('|'.join([PRINT, NAME]))  
  
for tok in generate_tokens(master_pat, 'printer'):  
    print(tok)  
  
# Outputs :  
# Token(type='PRINT', value='print')  
# Token(type='NAME', value='er')
```

For more advanced kinds of tokenizing, you may want to check out packages such as [PyParsing](#) or [PLY](#). An example involving [PLY](#) appears in the next recipe.

## 2.19. Writing a Simple Recursive Descent Parser

### Problem

You need to parse text according to a set of grammar rules and perform actions or build an abstract syntax tree representing the input. The grammar is small, so you'd prefer to just write the parser yourself as opposed to using some kind of framework.

### Solution

In this problem, we're focused on the problem of parsing text according to a particular grammar. In order to do this, you should probably start by having a formal specification of the grammar in the form of a BNF or EBNF. For example, a grammar for simple arithmetic expressions might look like this:

```
expr ::= expr + term  
      | expr - term  
      | term  
  
term ::= term * factor  
      | term / factor  
      | factor  
  
factor ::= ( expr )  
        | NUM
```

Or, alternatively, in EBNF form:

```

expr ::= term { (+|-) term }*

term ::= factor { (*|/) factor }*

factor ::= ( expr )
         |  NUM

```

In an EBNF, parts of a rule enclosed in { ... }\* are optional. The \* means zero or more repetitions (the same meaning as in a regular expression).

Now, if you're not familiar with the mechanics of working with a BNF, think of it as a specification of substitution or replacement rules where symbols on the left side can be replaced by the symbols on the right (or vice versa). Generally, what happens during parsing is that you try to match the input text to the grammar by making various substitutions and expansions using the BNF. To illustrate, suppose you are parsing an expression such as `3 + 4 * 5`. This expression would first need to be broken down into a token stream, using the techniques described in [Recipe 2.18](#). The result might be a sequence of tokens like this:

```
NUM + NUM * NUM
```

From there, parsing involves trying to match the grammar to input tokens by making substitutions:

```

expr
expr ::= term { (+|-) term }*
expr ::= factor { (*|/) factor }* { (+|-) term }*
expr ::= NUM { (*|/) factor }* { (+|-) term }*
expr ::= NUM { (+|-) term }*
expr ::= NUM + term { (+|-) term }*
expr ::= NUM + factor { (*|/) factor }* { (+|-) term }*
expr ::= NUM + NUM { (*|/) factor }* { (+|-) term }*
expr ::= NUM + NUM * factor { (*|/) factor }* { (+|-) term }*
expr ::= NUM + NUM * NUM { (*|/) factor }* { (+|-) term }*
expr ::= NUM + NUM * NUM { (+|-) term }*
expr ::= NUM + NUM * NUM

```

Following all of the substitution steps takes a bit of coffee, but they're driven by looking at the input and trying to match it to grammar rules. The first input token is a NUM, so substitutions first focus on matching that part. Once matched, attention moves to the next token of + and so on. Certain parts of the righthand side (e.g., { (\*|/) factor }\*) disappear when it's determined that they can't match the next token. In a successful parse, the entire righthand side is expanded completely to match the input token stream.

With all of the preceding background in place, here is a simple recipe that shows how to build a recursive descent expression evaluator:

```

import re
import collections

```



```

# Token specification
NUM    = r'(?P<NUM>\d+)'
PLUS   = r'(?P<PLUS>\+)'
MINUS  = r'(?P<MINUS>-)'
TIMES  = r'(?P<TIMES>\*)'
DIVIDE = r'(?P<DIVIDE>/)'
LPAREN = r'(?P<LPAREN>\(')
RPAREN = r'(?P<RPAREN>\))'
WS     = r'(?P<WS>\s+)'

master_pat = re.compile('|'.join([NUM, PLUS, MINUS, TIMES,
                                   DIVIDE, LPAREN, RPAREN, WS]))

# Tokenizer
Token = collections.namedtuple('Token', ['type', 'value'])

def generate_tokens(text):
    scanner = master_pat.scanner(text)
    for m in iter(scanner.match, None):
        tok = Token(m.lastgroup, m.group())
        if tok.type != 'WS':
            yield tok

# Parser
class ExpressionEvaluator:
    """
    Implementation of a recursive descent parser. Each method
    implements a single grammar rule. Use the ._accept() method
    to test and accept the current lookahead token. Use the ._expect()
    method to exactly match and discard the next token on the input
    (or raise a SyntaxError if it doesn't match).
    """

    def parse(self, text):
        self.tokens = generate_tokens(text)
        self.tok = None          # Last symbol consumed
        self.nexttok = None     # Next symbol tokenized
        self._advance()         # Load first lookahead token
        return self.expr()

    def _advance(self):
        'Advance one token ahead'
        self.tok, self.nexttok = self.nexttok, next(self.tokens, None)

    def _accept(self, toktype):
        'Test and consume the next token if it matches toktype'
        if self.nexttok and self.nexttok.type == toktype:
            self._advance()
            return True
        else:
            return False

```

```

def _expect(self, toktype):
    'Consume next token if it matches toktype or raise SyntaxError'
    if not self._accept(toktype):
        raise SyntaxError('Expected ' + toktype)

# Grammar rules follow

def expr(self):
    "expression ::= term { ('+'|'-') term }*"

    exprval = self.term()
    while self._accept('PLUS') or self._accept('MINUS'):
        op = self.tok.type
        right = self.term()
        if op == 'PLUS':
            exprval += right
        elif op == 'MINUS':
            exprval -= right
    return exprval

def term(self):
    "term ::= factor { ('*'|'/') factor }*"

    termval = self.factor()
    while self._accept('TIMES') or self._accept('DIVIDE'):
        op = self.tok.type
        right = self.factor()
        if op == 'TIMES':
            termval *= right
        elif op == 'DIVIDE':
            termval /= right
    return termval

def factor(self):
    "factor ::= NUM | ( expr )"

    if self._accept('NUM'):
        return int(self.tok.value)
    elif self._accept('LPAREN'):
        exprval = self.expr()
        self._expect('RPAREN')
        return exprval
    else:
        raise SyntaxError('Expected NUMBER or LPAREN')

```

Here is an example of using the ExpressionEvaluator class interactively:

```

>>> e = ExpressionEvaluator()
>>> e.parse('2')
2
>>> e.parse('2 + 3')
5

```

```

>>> e.parse('2 + 3 * 4')
14
>>> e.parse('2 + (3 + 4) * 5')
37
>>> e.parse('2 + (3 + * 4)')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "exprparse.py", line 40, in parse
    return self.expr()
  File "exprparse.py", line 67, in expr
    right = self.term()
  File "exprparse.py", line 77, in term
    termval = self.factor()
  File "exprparse.py", line 93, in factor
    exprval = self.expr()
  File "exprparse.py", line 67, in expr
    right = self.term()
  File "exprparse.py", line 77, in term
    termval = self.factor()
  File "exprparse.py", line 97, in factor
    raise SyntaxError("Expected NUMBER or LPAREN")
SyntaxError: Expected NUMBER or LPAREN
>>>

```

If you want to do something other than pure evaluation, you need to change the `ExpressionEvaluator` class to do something else. For example, here is an alternative implementation that constructs a simple parse tree:

```

class ExpressionTreeBuilder(ExpressionEvaluator):
    def expr(self):
        "expression ::= term { ('+'|'-') term }"

        exprval = self.term()
        while self._accept('PLUS') or self._accept('MINUS'):
            op = self.tok.type
            right = self.term()
            if op == 'PLUS':
                exprval = ('+', exprval, right)
            elif op == 'MINUS':
                exprval = ('-', exprval, right)
        return exprval

    def term(self):
        "term ::= factor { ('*'|'/') factor }"

        termval = self.factor()
        while self._accept('TIMES') or self._accept('DIVIDE'):
            op = self.tok.type
            right = self.factor()
            if op == 'TIMES':
                termval = ('*', termval, right)
            elif op == 'DIVIDE':

```

```

        termval = ('/', termval, right)
    return termval

def factor(self):
    'factor ::= NUM | ( expr )'

    if self._accept('NUM'):
        return int(self.tok.value)
    elif self._accept('LPAREN'):
        exprval = self.expr()
        self._expect('RPAREN')
        return exprval
    else:
        raise SyntaxError('Expected NUMBER or LPAREN')

```

The following example shows how it works:

```

>>> e = ExpressionTreeBuilder()
>>> e.parse('2 + 3')
('+', 2, 3)
>>> e.parse('2 + 3 * 4')
('+', 2, ('*', 3, 4))
>>> e.parse('2 + (3 + 4) * 5')
('+', 2, ('*', ('+', 3, 4), 5))
>>> e.parse('2 + 3 + 4')
('+', ('+', 2, 3), 4)
>>>

```

## Discussion

Parsing is a huge topic that generally occupies students for the first three weeks of a compilers course. If you are seeking background knowledge about grammars, parsing algorithms, and other information, a compilers book is where you should turn. Needless to say, all of that can't be repeated here.

Nevertheless, the overall idea of writing a recursive descent parser is generally simple. To start, you take every grammar rule and you turn it into a function or method. Thus, if your grammar looks like this:

```

expr ::= term { ('+'|'-') term }*

term ::= factor { ('*'|'/') factor }*

factor ::= '(' expr ')'
        | NUM

```

You start by turning it into a set of methods like this:

```

class ExpressionEvaluator:
    ...
    def expr(self):
        ...

```

```
def term(self):
    ...

def factor(self):
    ...
```

The task of each method is simple—it must walk from left to right over each part of the grammar rule, consuming tokens in the process. In a sense, the goal of the method is to either consume the rule or generate a syntax error if it gets stuck. To do this, the following implementation techniques are applied:

- If the next symbol in the rule is the name of another grammar rule (e.g., `term` or `factor`), you simply call the method with the same name. This is the “descent” part of the algorithm—control descends into another grammar rule. Sometimes rules will involve calls to methods that are already executing (e.g., the call to `expr` in the `factor ::= '(' expr ')'` rule). This is the “recursive” part of the algorithm.
- If the next symbol in the rule has to be a specific symbol (e.g., `(`), you look at the next token and check for an exact match. If it doesn’t match, it’s a syntax error. The `_expect()` method in this recipe is used to perform these steps.
- If the next symbol in the rule could be a few possible choices (e.g., `+` or `-`), you have to check the next token for each possibility and advance only if a match is made. This is the purpose of the `_accept()` method in this recipe. It’s kind of like a weaker version of the `_expect()` method in that it will advance if a match is made, but if not, it simply backs off without raising an error (thus allowing further checks to be made).
- For grammar rules where there are repeated parts (e.g., such as in the rule `expr ::= term { ('+' | '-') term }*`), the repetition gets implemented by a `while` loop. The body of the loop will generally collect or process all of the repeated items until no more are found.
- Once an entire grammar rule has been consumed, each method returns some kind of result back to the caller. This is how values propagate during parsing. For example, in the expression evaluator, return values will represent partial results of the expression being parsed. Eventually they all get combined together in the topmost grammar rule method that executes.

Although a simple example has been shown, recursive descent parsers can be used to implement rather complicated parsers. For example, Python code itself is interpreted by a recursive descent parser. If you’re so inclined, you can look at the underlying grammar by inspecting the file *Grammar/Grammar* in the Python source. That said, there are still numerous pitfalls and limitations with making a parser by hand.

One such limitation of recursive descent parsers is that they can't be written for grammar rules involving any kind of left recursion. For example, suppose you need to translate a rule like this:

```
items ::= items ',' item
        | item
```

To do it, you might try to use the `items()` method like this:

```
def items(self):
    itemsval = self.items()
    if itemsval and self._accept(','):
        itemsval.append(self.item())
    else:
        itemsval = [ self.item() ]
```

The only problem is that it doesn't work. In fact, it blows up with an infinite recursion error.

You can also run into tricky issues concerning the grammar rules themselves. For example, you might have wondered whether or not expressions could have been described by this more simple grammar:

```
expr ::= factor { ('+'| '-'| '*'| '/') factor }*

factor ::= '(' expression ')'
        | NUM
```

This grammar technically “works,” but it doesn't observe the standard arithmetic rules concerning order of evaluation. For example, the expression “3 + 4 \* 5” would get evaluated as “35” instead of the expected result of “23.” The use of separate “expr” and “term” rules is there to make evaluation work correctly.

For really complicated grammars, you are often better off using parsing tools such as **PyParsing** or **PLY**. This is what the expression evaluator code looks like using PLY:

```
from ply.lex import lex
from ply.yacc import yacc

# Token list
tokens = [ 'NUM', 'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'LPAREN', 'RPAREN' ]

# Ignored characters
t_ignore = ' \t\n'

# Token specifications (as regexs)
t_PLUS = r'\+'
t_MINUS = r'\-'
t_TIMES = r'\*'
t_DIVIDE = r'\/'
t_LPAREN = r'\('
```

```

t_RPAREN = r'\)'

# Token processing functions
def t_NUM(t):
    r'\d+'
    t.value = int(t.value)
    return t

# Error handler
def t_error(t):
    print('Bad character: {!r}'.format(t.value[0]))
    t.skip(1)

# Build the lexer
lexer = lex()

# Grammar rules and handler functions
def p_expr(p):
    '''
    expr : expr PLUS term
         | expr MINUS term
    '''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]

def p_expr_term(p):
    '''
    expr : term
    '''
    p[0] = p[1]

def p_term(p):
    '''
    term : term TIMES factor
         | term DIVIDE factor
    '''
    if p[2] == '*':
        p[0] = p[1] * p[3]
    elif p[2] == '/':
        p[0] = p[1] / p[3]

def p_term_factor(p):
    '''
    term : factor
    '''
    p[0] = p[1]

def p_factor(p):
    '''
    factor : NUM

```

```

'''
p[0] = p[1]

def p_factor_group(p):
'''
    factor : LPAREN expr RPAREN
    '''
    p[0] = p[2]

def p_error(p):
    print('Syntax error')

parser = yacc()

```

In this code, you'll find that everything is specified at a much higher level. You simply write regular expressions for the tokens and high-level handling functions that execute when various grammar rules are matched. The actual mechanics of running the parser, accepting tokens, and so forth is implemented entirely by the library.

Here is an example of how the resulting parser object gets used:

```

>>> parser.parse('2')
2
>>> parser.parse('2+3')
5
>>> parser.parse('2+(3+4)*5')
37
>>>

```

If you need a bit more excitement in your programming, writing parsers and compilers can be a fun project. Again, a compilers textbook will have a lot of low-level details underlying theory. However, many fine resources can also be found online. Python's own `ast` module is also worth a look.

## 2.20. Performing Text Operations on Byte Strings

### Problem

You want to perform common text operations (e.g., stripping, searching, and replacement) on byte strings.

### Solution

Byte strings already support most of the same built-in operations as text strings. For example:



```

>>> data = b'Hello World'
>>> data[0:5]
b'Hello'
>>> data.startswith(b'Hello')
True
>>> data.split()
[b'Hello', b'World']
>>> data.replace(b'Hello', b'Hello Cruel')
b'Hello Cruel World'
>>>

```

Such operations also work with byte arrays. For example:

```

>>> data = bytearray(b'Hello World')
>>> data[0:5]
bytearray(b'Hello')
>>> data.startswith(b'Hello')
True
>>> data.split()
[bytearray(b'Hello'), bytearray(b'World')]
>>> data.replace(b'Hello', b'Hello Cruel')
bytearray(b'Hello Cruel World')
>>>

```

You can apply regular expression pattern matching to byte strings, but the patterns themselves need to be specified as bytes. For example:

```

>>>
>>> data = b'FOO:BAR,SPAM'
>>> import re
>>> re.split('[:,]',data)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.3/re.py", line 191, in split
    return _compile(pattern, flags).split(string, maxsplit)
TypeError: can't use a string pattern on a bytes-like object

>>> re.split(b'[:,]',data)      # Notice: pattern as bytes
[b'FOO', b'BAR', b'SPAM']
>>>

```

## Discussion

For the most part, almost all of the operations available on text strings will work on byte strings. However, there are a few notable differences to be aware of. First, indexing of byte strings produces integers, not individual characters. For example:

```

>>> a = 'Hello World'      # Text string
>>> a[0]
'H'
>>> a[1]
'e'
>>> b = b'Hello World'     # Byte string

```

```
>>> b[0]
72
>>> b[1]
101
>>>
```

This difference in semantics can affect programs that try to process byte-oriented data on a character-by-character basis.

Second, byte strings don't provide a nice string representation and don't print cleanly unless first decoded into a text string. For example:

```
>>> s = b'Hello World'
>>> print(s)
b'Hello World'          # Observe b'...'
>>> print(s.decode('ascii'))
Hello World
>>>
```

Similarly, there are no string formatting operations available to byte strings.

```
>>> b'%10s %10d %10.2f' % (b'ACME', 100, 490.1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for %: 'bytes' and 'tuple'

>>> b'{} {} {}'.format(b'ACME', 100, 490.1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'bytes' object has no attribute 'format'
>>>
```

If you want to do any kind of formatting applied to byte strings, it should be done using normal text strings and encoding. For example:

```
>>> '{:10s} {:10d} {:10.2f}'.format('ACME', 100, 490.1).encode('ascii')
b'ACME          100      490.10'
>>>
```

Finally, you need to be aware that using a byte string can change the semantics of certain operations—especially those related to the filesystem. For example, if you supply a filename encoded as bytes instead of a text string, it usually disables filename encoding/decoding. For example:

```
>>> # Write a UTF-8 filename
>>> with open('jalape\xfb1o.txt', 'w') as f:
...     f.write('spicy')
...

>>> # Get a directory listing
>>> import os
>>> os.listdir('.')          # Text string (names are decoded)
['jalape\u00f1o.txt']
```

```
>>> os.listdir(b'.')          # Byte string (names left as bytes)
[b'jalapen\xcc\x83o.txt']
>>>
```

Notice in the last part of this example how giving a byte string as the directory name caused the resulting filenames to be returned as undecoded bytes. The filename shown in the directory listing contains raw UTF-8 encoding. See [Recipe 5.15](#) for some related issues concerning filenames.

As a final comment, some programmers might be inclined to use byte strings as an alternative to text strings due to a possible performance improvement. Although it's true that manipulating bytes tends to be slightly more efficient than text (due to the inherent overhead related to Unicode), doing so usually leads to very messy and nonidiomatic code. You'll often find that byte strings don't play well with a lot of other parts of Python, and that you end up having to perform all sorts of manual encoding/decoding operations yourself to get things to work right. Frankly, if you're working with text, use normal text strings in your program, not byte strings.



---

# Numbers, Dates, and Times

Performing mathematical calculations with integers and floating-point numbers is easy in Python. However, if you need to perform calculations with fractions, arrays, or dates and times, a bit more work is required. The focus of this chapter is on such topics.

## 3.1. Rounding Numerical Values

### Problem

You want to round a floating-point number to a fixed number of decimal places.

### Solution

For simple rounding, use the built-in `round(value, ndigits)` function. For example:

```
>>> round(1.23, 1)
1.2
>>> round(1.27, 1)
1.3
>>> round(-1.27, 1)
-1.3
>>> round(1.25361, 3)
1.254
>>>
```

When a value is exactly halfway between two choices, the behavior of `round` is to round to the nearest even digit. That is, values such as 1.5 or 2.5 both get rounded to 2.

The number of digits given to `round()` can be negative, in which case rounding takes place for tens, hundreds, thousands, and so on. For example:

```
>>> a = 1627731
>>> round(a, -1)
1627730
```

```
>>> round(a, -2)
1627700
>>> round(a, -3)
1628000
>>>
```

## Discussion

Don't confuse rounding with formatting a value for output. If your goal is simply to output a numerical value with a certain number of decimal places, you don't typically need to use `round()`. Instead, just specify the desired precision when formatting. For example:

```
>>> x = 1.23456
>>> format(x, '0.2f')
'1.23'
>>> format(x, '0.3f')
'1.235'
>>> 'value is {:.03f}'.format(x)
'value is 1.235'
>>>
```

Also, resist the urge to round floating-point numbers to “fix” perceived accuracy problems. For example, you might be inclined to do this:

```
>>> a = 2.1
>>> b = 4.2
>>> c = a + b
>>> c
6.300000000000001
>>> c = round(c, 2)      # "Fix" result (???)
>>> c
6.3
>>>
```

For most applications involving floating point, it's simply not necessary (or recommended) to do this. Although there are small errors introduced into calculations, the behavior of those errors are understood and tolerated. If avoiding such errors is important (e.g., in financial applications, perhaps), consider the use of the `decimal` module, which is discussed in the next recipe.

## 3.2. Performing Accurate Decimal Calculations

### Problem

You need to perform accurate calculations with decimal numbers, and don't want the small errors that naturally occur with floats.

## Solution

A well-known issue with floating-point numbers is that they can't accurately represent all base-10 decimals. Moreover, even simple mathematical calculations introduce small errors. For example:

```
>>> a = 4.2
>>> b = 2.1
>>> a + b
6.300000000000001
>>> (a + b) == 6.3
False
>>>
```

These errors are a “feature” of the underlying CPU and the IEEE 754 arithmetic performed by its floating-point unit. Since Python's float data type stores data using the native representation, there's nothing you can do to avoid such errors if you write your code using float instances.

If you want more accuracy (and are willing to give up some performance), you can use the decimal module:

```
>>> from decimal import Decimal
>>> a = Decimal('4.2')
>>> b = Decimal('2.1')
>>> a + b
Decimal('6.3')
>>> print(a + b)
6.3
>>> (a + b) == Decimal('6.3')
True
>>>
```

At first glance, it might look a little weird (i.e., specifying numbers as strings). However, Decimal objects work in every way that you would expect them to (supporting all of the usual math operations, etc.). If you print them or use them in string formatting functions, they look like normal numbers.

A major feature of decimal is that it allows you to control different aspects of calculations, including number of digits and rounding. To do this, you create a local context and change its settings. For example:

```
>>> from decimal import localcontext
>>> a = Decimal('1.3')
>>> b = Decimal('1.7')
>>> print(a / b)
0.7647058823529411764705882353
>>> with localcontext() as ctx:
...     ctx.prec = 3
...     print(a / b)
... 
```

```

0.765
>>> with localcontext() as ctx:
...     ctx.prec = 50
...     print(a / b)
...
0.76470588235294117647058823529411764705882352941176
>>>

```

## Discussion

The `decimal` module implements IBM’s “General Decimal Arithmetic Specification.” Needless to say, there are a huge number of configuration options that are beyond the scope of this book.

Newcomers to Python might be inclined to use the `decimal` module to work around perceived accuracy problems with the `float` data type. However, it’s really important to understand your application domain. If you’re working with science or engineering problems, computer graphics, or most things of a scientific nature, it’s simply more common to use the normal floating-point type. For one, very few things in the real world are measured to the 17 digits of accuracy that floats provide. Thus, tiny errors introduced in calculations just don’t matter. Second, the performance of native floats is significantly faster—something that’s important if you’re performing a large number of calculations.

That said, you can’t ignore the errors completely. Mathematicians have spent a lot of time studying various algorithms, and some handle errors better than others. You also have to be a little careful with effects due to things such as subtractive cancellation and adding large and small numbers together. For example:

```

>>> nums = [1.23e+18, 1, -1.23e+18]
>>> sum(nums)      # Notice how 1 disappears
0.0
>>>

```

This latter example can be addressed by using a more accurate implementation in `math.fsum()`:

```

>>> import math
>>> math.fsum(nums)
1.0
>>>

```

However, for other algorithms, you really need to study the algorithm and understand its error propagation properties.

All of this said, the main use of the `decimal` module is in programs involving things such as finance. In such programs, it is extremely annoying to have small errors creep into the calculation. Thus, `decimal` provides a way to avoid that. It is also common to encounter `Decimal` objects when Python interfaces with databases—again, especially when accessing financial data.



## 3.3. Formatting Numbers for Output

### Problem

You need to format a number for output, controlling the number of digits, alignment, inclusion of a thousands separator, and other details.

### Solution

To format a single number for output, use the built-in `format()` function. For example:

```
>>> x = 1234.56789

>>> # Two decimal places of accuracy
>>> format(x, '0.2f')
'1234.57'

>>> # Right justified in 10 chars, one-digit accuracy
>>> format(x, '>10.1f')
'    1234.6'

>>> # Left justified
>>> format(x, '<10.1f')
'1234.6    '

>>> # Centered
>>> format(x, '^10.1f')
'  1234.6  '

>>> # Inclusion of thousands separator
>>> format(x, ',')
'1,234.56789'
>>> format(x, '0,.1f')
'1,234.6'
>>>
```

If you want to use exponential notation, change the `f` to an `e` or `E`, depending on the case you want used for the exponential specifier. For example:

```
>>> format(x, 'e')
'1.234568e+03'
>>> format(x, '0.2E')
'1.23E+03'
>>>
```

The general form of the width and precision in both cases is `'[<>^]?width[,]?(.digits)?'` where `width` and `digits` are integers and `?` signifies optional parts. The same format codes are also used in the `.format()` method of strings. For example:

```
>>> 'The value is {:0,.2f}'.format(x)
'The value is 1,234.57'
>>>
```

## Discussion

Formatting numbers for output is usually straightforward. The technique shown works for both floating-point numbers and `Decimal` numbers in the `decimal` module.

When the number of digits is restricted, values are rounded away according to the same rules of the `round()` function. For example:

```
>>> x
1234.56789
>>> format(x, '0.1f')
'1234.6'
>>> format(-x, '0.1f')
'-1234.6'
>>>
```

Formatting of values with a thousands separator is not locale aware. If you need to take that into account, you might investigate functions in the `locale` module. You can also swap separator characters using the `translate()` method of strings. For example:

```
>>> swap_separators = { ord('.'): ',', ord(','): '.' }
>>> format(x, ',').translate(swap_separators)
'1.234,56789'
>>>
```

In a lot of Python code, numbers are formatted using the `%` operator. For example:

```
>>> '%0.2f' % x
'1234.57'
>>> '%10.1f' % x
'      1234.6'
>>> '%-10.1f' % x
'1234.6      '
>>>
```

This formatting is still acceptable, but less powerful than the more modern `format()` method. For example, some features (e.g., adding thousands separators) aren't supported when using the `%` operator to format numbers.

## 3.4. Working with Binary, Octal, and Hexadecimal Integers

### Problem

You need to convert or output integers represented by binary, octal, or hexadecimal digits.

### Solution

To convert an integer into a binary, octal, or hexadecimal text string, use the `bin()`, `oct()`, or `hex()` functions, respectively:

```
>>> x = 1234
>>> bin(x)
'0b10011010010'
>>> oct(x)
'0o2322'
>>> hex(x)
'0x4d2'
>>>
```

Alternatively, you can use the `format()` function if you don't want the `0b`, `0o`, or `0x` prefixes to appear. For example:

```
>>> format(x, 'b')
'10011010010'
>>> format(x, 'o')
'2322'
>>> format(x, 'x')
'4d2'
>>>
```

Integers are signed, so if you are working with negative numbers, the output will also include a sign. For example:

```
>>> x = -1234
>>> format(x, 'b')
'-10011010010'
>>> format(x, 'x')
'-4d2'
>>>
```

If you need to produce an unsigned value instead, you'll need to add in the maximum value to set the bit length. For example, to show a 32-bit value, use the following:

```
>>> x = -1234
>>> format(2**32 + x, 'b')
'11111111111111111111111111111110'
>>> format(2**32 + x, 'x')
```

```
'fffffb2e'
>>>
```

To convert integer strings in different bases, simply use the `int()` function with an appropriate base. For example:

```
>>> int('4d2', 16)
1234
>>> int('10011010010', 2)
1234
>>>
```

## Discussion

For the most part, working with binary, octal, and hexadecimal integers is straightforward. Just remember that these conversions only pertain to the conversion of integers to and from a textual representation. Under the covers, there's just one integer type.

Finally, there is one caution for programmers who use octal. The Python syntax for specifying octal values is slightly different than many other languages. For example, if you try something like this, you'll get a syntax error:

```
>>> import os
>>> os.chmod('script.py', 0755)
File "<stdin>", line 1
    os.chmod('script.py', 0755)
                             ^
SyntaxError: invalid token
>>>
```

Make sure you prefix the octal value with `0o`, as shown here:

```
>>> os.chmod('script.py', 0o755)
>>>
```

## 3.5. Packing and Unpacking Large Integers from Bytes

### Problem

You have a byte string and you need to unpack it into an integer value. Alternatively, you need to convert a large integer back into a byte string.

### Solution

Suppose your program needs to work with a 16-element byte string that holds a 128-bit integer value. For example:

```
data = b'\x00\x124V\x00x\x90\xab\x00\xcd\xef\x01\x00#\x004'
```

To interpret the bytes as an integer, use `int.from_bytes()`, and specify the byte ordering like this:

```
>>> len(data)
16
>>> int.from_bytes(data, 'little')
69120565665751139577663547927094891008
>>> int.from_bytes(data, 'big')
94522842520747284487117727783387188
>>>
```

To convert a large integer value back into a byte string, use the `int.to_bytes()` method, specifying the number of bytes and the byte order. For example:

```
>>> x = 94522842520747284487117727783387188
>>> x.to_bytes(16, 'big')
b'\x00\x124V\x00x\x90\xab\x00\xcd\xef\x01\x00#\x004'
>>> x.to_bytes(16, 'little')
b'4\x00#\x00\x01\xef\xcd\x00\xab\x90x\x00V4\x12\x00'
>>>
```

## Discussion

Converting large integer values to and from byte strings is not a common operation. However, it sometimes arises in certain application domains, such as cryptography or networking. For instance, IPv6 network addresses are represented as 128-bit integers. If you are writing code that needs to pull such values out of a data record, you might face this problem.

As an alternative to this recipe, you might be inclined to unpack values using the `struct` module, as described in [Recipe 6.11](#). This works, but the size of integers that can be unpacked with `struct` is limited. Thus, you would need to unpack multiple values and combine them to create the final value. For example:

```
>>> data
b'\x00\x124V\x00x\x90\xab\x00\xcd\xef\x01\x00#\x004'
>>> import struct
>>> hi, lo = struct.unpack('>QQ', data)
>>> (hi << 64) + lo
94522842520747284487117727783387188
>>>
```

The specification of the byte order (`little` or `big`) just indicates whether the bytes that make up the integer value are listed from the least to most significant or the other way around. This is easy to view using a carefully crafted hexadecimal value:

```
>>> x = 0x01020304
>>> x.to_bytes(4, 'big')
b'\x01\x02\x03\x04'
>>> x.to_bytes(4, 'little')
b'\x04\x03\x02\x01'
```

```
b'\x04\x03\x02\x01'
>>>
```

If you try to pack an integer into a byte string, but it won't fit, you'll get an error. You can use the `int.bit_length()` method to determine how many bits are required to store a value if needed:

```
>>> x = 523 ** 23
>>> x
33538130011366187510753685271401905616035565533978849017944067
>>> x.to_bytes(16, 'little')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: int too big to convert
>>> x.bit_length()
208
>>> nbytes, rem = divmod(x.bit_length(), 8)
>>> if rem:
...     nbytes += 1
...
>>>
>>> x.to_bytes(nbytes, 'little')
b'\x03X\xf1\x82iT\x96\xac\xc7c\x16\xf3\xb9\xcf...\xd0'
>>>
```

## 3.6. Performing Complex-Valued Math

### Problem

Your code for interacting with the latest web authentication scheme has encountered a singularity and your only solution is to go around it in the complex plane. Or maybe you just need to perform some calculations using complex numbers.

### Solution

Complex numbers can be specified using the `complex(real, imag)` function or by floating-point numbers with a `j` suffix. For example:

```
>>> a = complex(2, 4)
>>> b = 3 - 5j
>>> a
(2+4j)
>>> b
(3-5j)
>>>
```

The real, imaginary, and conjugate values are easy to obtain, as shown here:

```
>>> a.real
2.0
```

```
>>> a.imag
4.0
>>> a.conjugate()
(2-4j)
>>>
```

In addition, all of the usual mathematical operators work:

```
>>> a + b
(5-1j)
>>> a * b
(26+2j)
>>> a / b
(-0.4117647058823529+0.6470588235294118j)
>>> abs(a)
4.47213595499958
>>>
```

To perform additional complex-valued functions such as sines, cosines, or square roots, use the `cmath` module:

```
>>> import cmath
>>> cmath.sin(a)
(24.83130584894638-11.356612711218174j)
>>> cmath.cos(a)
(-11.36423470640106-24.814651485634187j)
>>> cmath.exp(a)
(-4.829809383269385-5.5920560936409816j)
>>>
```

## Discussion

Most of Python's math-related modules are aware of complex values. For example, if you use `numpy`, it is straightforward to make arrays of complex values and perform operations on them:

```
>>> import numpy as np
>>> a = np.array([2+3j, 4+5j, 6-7j, 8+9j])
>>> a
array([ 2.+3.j,  4.+5.j,  6.-7.j,  8.+9.j])
>>> a + 2
array([ 4.+3.j,  6.+5.j,  8.-7.j, 10.+9.j])
>>> np.sin(a)
array([  9.15449915 -4.16890696j, -56.16227422 -48.50245524j,
        -153.20827755-526.47684926j,  4008.42651446-589.49948373j])
>>>
```

Python's standard mathematical functions do not produce complex values by default, so it is unlikely that such a value would accidentally show up in your code. For example:

```
>>> import math
>>> math.sqrt(-1)
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
ValueError: math domain error
>>>
```

If you want complex numbers to be produced as a result, you have to explicitly use `cmath` or declare the use of a complex type in libraries that know about them. For example:

```
>>> import cmath
>>> cmath.sqrt(-1)
1j
>>>
```

## 3.7. Working with Infinity and NaNs

### Problem

You need to create or test for the floating-point values of infinity, negative infinity, or NaN (not a number).

### Solution

Python has no special syntax to represent these special floating-point values, but they can be created using `float()`. For example:

```
>>> a = float('inf')
>>> b = float('-inf')
>>> c = float('nan')
>>> a
inf
>>> b
-inf
>>> c
nan
>>>
```

To test for the presence of these values, use the `math.isinf()` and `math.isnan()` functions. For example:

```
>>> math.isinf(a)
True
>>> math.isnan(c)
True
>>>
```

### Discussion

For more detailed information about these special floating-point values, you should refer to the IEEE 754 specification. However, there are a few tricky details to be aware of, especially related to comparisons and operators.



Infinite values will propagate in calculations in a mathematical manner. For example:

```
>>> a = float('inf')
>>> a + 45
inf
>>> a * 10
inf
>>> 10 / a
0.0
>>>
```

However, certain operations are undefined and will result in a NaN result. For example:

```
>>> a = float('inf')
>>> a/a
nan
>>> b = float('-inf')
>>> a + b
nan
>>>
```

NaN values propagate through all operations without raising an exception. For example:

```
>>> c = float('nan')
>>> c + 23
nan
>>> c / 2
nan
>>> c * 2
nan
>>> math.sqrt(c)
nan
>>>
```

A subtle feature of NaN values is that they never compare as equal. For example:

```
>>> c = float('nan')
>>> d = float('nan')
>>> c == d
False
>>> c is d
False
>>>
```

Because of this, the only safe way to test for a NaN value is to use `math.isnan()`, as shown in this recipe.

Sometimes programmers want to change Python's behavior to raise exceptions when operations result in an infinite or NaN result. The `fpectl` module can be used to adjust this behavior, but it is not enabled in a standard Python build, it's platform-dependent, and really only intended for expert-level programmers. See [the online Python documentation](#) for further details.

## 3.8. Calculating with Fractions

### Problem

You have entered a time machine and suddenly find yourself working on elementary-level homework problems involving fractions. Or perhaps you're writing code to make calculations involving measurements made in your wood shop.

### Solution

The fractions module can be used to perform mathematical calculations involving fractions. For example:

```
>>> from fractions import Fraction
>>> a = Fraction(5, 4)
>>> b = Fraction(7, 16)
>>> print(a + b)
27/16
>>> print(a * b)
35/64

>>> # Getting numerator/denominator
>>> c = a * b
>>> c.numerator
35
>>> c.denominator
64

>>> # Converting to a float
>>> float(c)
0.546875

>>> # Limiting the denominator of a value
>>> print(c.limit_denominator(8))
4/7

>>> # Converting a float to a fraction
>>> x = 3.75
>>> y = Fraction(*x.as_integer_ratio())
>>> y
Fraction(15, 4)
>>>
```

### Discussion

Calculating with fractions doesn't arise often in most programs, but there are situations where it might make sense to use them. For example, allowing a program to accept units of measurement in fractions and performing calculations with them in that form might alleviate the need for a user to manually make conversions to decimals or floats.

## 3.9. Calculating with Large Numerical Arrays

### Problem

You need to perform calculations on large numerical datasets, such as arrays or grids.

### Solution

For any heavy computation involving arrays, use the **NumPy library**. The major feature of NumPy is that it gives Python an array object that is much more efficient and better suited for mathematical calculation than a standard Python list. Here is a short example illustrating important behavioral differences between lists and NumPy arrays:

```
>>> # Python lists
>>> x = [1, 2, 3, 4]
>>> y = [5, 6, 7, 8]
>>> x * 2
[1, 2, 3, 4, 1, 2, 3, 4]
>>> x + 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list
>>> x + y
[1, 2, 3, 4, 5, 6, 7, 8]

>>> # Numpy arrays
>>> import numpy as np
>>> ax = np.array([1, 2, 3, 4])
>>> ay = np.array([5, 6, 7, 8])
>>> ax * 2
array([2, 4, 6, 8])
>>> ax + 10
array([11, 12, 13, 14])
>>> ax + ay
array([ 6,  8, 10, 12])
>>> ax * ay
array([ 5, 12, 21, 32])
>>>
```

As you can see, basic mathematical operations involving arrays behave differently. Specifically, scalar operations (e.g., `ax * 2` or `ax + 10`) apply the operation on an element-by-element basis. In addition, performing math operations when both operands are arrays applies the operation to all elements and produces a new array.

The fact that math operations apply to all of the elements simultaneously makes it very easy and fast to compute functions across an entire array. For example, if you want to compute the value of a polynomial:

```
>>> def f(x):
...     return 3*x**2 - 2*x + 7
```

```
...
>>> f(ax)
array([ 8, 15, 28, 47])
>>>
```

NumPy provides a collection of “universal functions” that also allow for array operations. These are replacements for similar functions normally found in the `math` module. For example:

```
>>> np.sqrt(ax)
array([ 1.          ,  1.41421356,  1.73205081,  2.          ])
>>> np.cos(ax)
array([ 0.54030231, -0.41614684, -0.9899925 , -0.65364362])
>>>
```

Using universal functions can be hundreds of times faster than looping over the array elements one at a time and performing calculations using functions in the `math` module. Thus, you should prefer their use whenever possible.

Under the covers, NumPy arrays are allocated in the same manner as in C or Fortran. Namely, they are large, contiguous memory regions consisting of a homogenous data type. Because of this, it’s possible to make arrays much larger than anything you would normally put into a Python list. For example, if you want to make a two-dimensional grid of 10,000 by 10,000 floats, it’s not an issue:

```
>>> grid = np.zeros(shape=(10000,10000), dtype=float)
>>> grid
array([[ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       ...,
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.]])
>>>
```

All of the usual operations still apply to all of the elements simultaneously:

```
>>> grid += 10
>>> grid
array([[ 10.,  10.,  10., ...,  10.,  10.,  10.],
       [ 10.,  10.,  10., ...,  10.,  10.,  10.],
       [ 10.,  10.,  10., ...,  10.,  10.,  10.],
       ...,
       [ 10.,  10.,  10., ...,  10.,  10.,  10.],
       [ 10.,  10.,  10., ...,  10.,  10.,  10.],
       [ 10.,  10.,  10., ...,  10.,  10.,  10.]])
>>> np.sin(grid)
array([[ -0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111],
       [ -0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111],
       [ -0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111],
       ...,
       [ -0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111],
       [ -0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111],
       [ -0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
        -0.54402111, -0.54402111]])
```

```

[-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
 -0.54402111, -0.54402111],
...,
[-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
 -0.54402111, -0.54402111],
[-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
 -0.54402111, -0.54402111],
[-0.54402111, -0.54402111, -0.54402111, ..., -0.54402111,
 -0.54402111, -0.54402111]])
>>>

```

One extremely notable aspect of NumPy is the manner in which it extends Python's list indexing functionality—especially with multidimensional arrays. To illustrate, make a simple two-dimensional array and try some experiments:

```

>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
>>> a
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])

>>> # Select row 1
>>> a[1]
array([5, 6, 7, 8])

>>> # Select column 1
>>> a[:,1]
array([ 2,  6, 10])

>>> # Select a subregion and change it
>>> a[1:3, 1:3]
array([[ 6,  7],
       [10, 11]])
>>> a[1:3, 1:3] += 10
>>> a
array([[ 1,  2,  3,  4],
       [ 5, 16, 17,  8],
       [ 9, 20, 21, 12]])

>>> # Broadcast a row vector across an operation on all rows
>>> a + [100, 101, 102, 103]
array([[101, 103, 105, 107],
       [105, 117, 119, 111],
       [109, 121, 123, 115]])

>>> a
array([[ 1,  2,  3,  4],
       [ 5, 16, 17,  8],
       [ 9, 20, 21, 12]])

>>> # Conditional assignment on an array
>>> np.where(a < 10, a, 10)
array([[ 1,  2,  3,  4],

```

```
[ 5, 10, 10,  8],
 [ 9, 10, 10, 10]])
>>>
```

## Discussion

NumPy is the foundation for a huge number of science and engineering libraries in Python. It is also one of the largest and most complicated modules in widespread use. That said, it's still possible to accomplish useful things with NumPy by starting with simple examples and playing around.

One note about usage is that it is relatively common to use the statement `import numpy as np`, as shown in the solution. This simply shortens the name to something that's more convenient to type over and over again in your program.

For more information, you definitely need to visit <http://www.numpy.org>.

## 3.10. Performing Matrix and Linear Algebra Calculations

### Problem

You need to perform matrix and linear algebra operations, such as matrix multiplication, finding determinants, solving linear equations, and so on.

### Solution

The **NumPy library** has a `matrix` object that can be used for this purpose. Matrices are somewhat similar to the array objects described in [Recipe 3.9](#), but follow linear algebra rules for computation. Here is an example that illustrates a few essential features:

```
>>> import numpy as np
>>> m = np.matrix([[1,-2,3],[0,4,5],[7,8,-9]])
>>> m
matrix([[ 1, -2,  3],
        [ 0,  4,  5],
        [ 7,  8, -9]])

>>> # Return transpose
>>> m.T
matrix([[ 1,  0,  7],
        [-2,  4,  8],
        [ 3,  5, -9]])

>>> # Return inverse
>>> m.I
matrix([[ 0.33043478, -0.02608696,  0.09565217],
        [-0.15217391,  0.13043478,  0.02173913],
        [ 0.12173913,  0.09565217, -0.0173913 ]])
```

```

>>> # Create a vector and multiply
>>> v = np.matrix([[2],[3],[4]])
>>> v
matrix([[2],
        [3],
        [4]])
>>> m * v
matrix([[ 8],
        [32],
        [ 2]])
>>>

```

More operations can be found in the `numpy.linalg` subpackage. For example:

```

>>> import numpy.linalg

>>> # Determinant
>>> numpy.linalg.det(m)
-229.99999999999983

>>> # Eigenvalues
>>> numpy.linalg.eigvals(m)
array([-13.11474312,  2.75956154,  6.35518158])

>>> # Solve for x in mx = v
>>> x = numpy.linalg.solve(m, v)
>>> x
matrix([[ 0.96521739],
        [ 0.17391304],
        [ 0.46086957]])
>>> m * x
matrix([[ 2.],
        [ 3.],
        [ 4.]])
>>> v
matrix([[2],
        [3],
        [4]])
>>>

```

## Discussion

Linear algebra is obviously a huge topic that's far beyond the scope of this cookbook. However, if you need to manipulate matrices and vectors, NumPy is a good starting point. Visit <http://www.numpy.org> for more detailed information.

## 3.11. Picking Things at Random

### Problem

You want to pick random items out of a sequence or generate random numbers.

### Solution

The `random` module has various functions for random numbers and picking random items. For example, to pick a random item out of a sequence, use `random.choice()`:

```
>>> import random
>>> values = [1, 2, 3, 4, 5, 6]
>>> random.choice(values)
2
>>> random.choice(values)
3
>>> random.choice(values)
1
>>> random.choice(values)
4
>>> random.choice(values)
6
>>>
```

To take a sampling of `N` items where selected items are removed from further consideration, use `random.sample()` instead:

```
>>> random.sample(values, 2)
[6, 2]
>>> random.sample(values, 2)
[4, 3]
>>> random.sample(values, 3)
[4, 3, 1]
>>> random.sample(values, 3)
[5, 4, 1]
>>>
```

If you simply want to shuffle items in a sequence in place, use `random.shuffle()`:

```
>>> random.shuffle(values)
>>> values
[2, 4, 6, 5, 3, 1]
>>> random.shuffle(values)
>>> values
[3, 5, 2, 1, 6, 4]
>>>
```

To produce random integers, use `random.randint()`:

```
>>> random.randint(0, 10)
2
```



```

>>> random.randint(0,10)
5
>>> random.randint(0,10)
0
>>> random.randint(0,10)
7
>>> random.randint(0,10)
10
>>> random.randint(0,10)
3
>>>

```

To produce uniform floating-point values in the range 0 to 1, use `random.random()`:

```

>>> random.random()
0.9406677561675867
>>> random.random()
0.133129581343897
>>> random.random()
0.4144991136919316
>>>

```

To get N random-bits expressed as an integer, use `random.getrandbits()`:

```

>>> random.getrandbits(200)
335837000776573622800628485064121869519521710558559406913275
>>>

```

## Discussion

The `random` module computes random numbers using the Mersenne Twister algorithm. This is a deterministic algorithm, but you can alter the initial seed by using the `random.seed()` function. For example:

```

random.seed()           # Seed based on system time or os.urandom()
random.seed(12345)      # Seed based on integer given
random.seed(b'bytedata') # Seed based on byte data

```

In addition to the functionality shown, `random()` includes functions for uniform, Gaussian, and other probability distributions. For example, `random.uniform()` computes uniformly distributed numbers, and `random.gauss()` computes normally distributed numbers. Consult the documentation for information on other supported distributions.

Functions in `random()` should not be used in programs related to cryptography. If you need such functionality, consider using functions in the `ssl` module instead. For example, `ssl.RAND_bytes()` can be used to generate a cryptographically secure sequence of random bytes.

## 3.12. Converting Days to Seconds, and Other Basic Time Conversions

### Problem

You have code that needs to perform simple time conversions, like days to seconds, hours to minutes, and so on.

### Solution

To perform conversions and arithmetic involving different units of time, use the `datetime` module. For example, to represent an interval of time, create a `timedelta` instance, like this:

```
>>> from datetime import timedelta
>>> a = timedelta(days=2, hours=6)
>>> b = timedelta(hours=4.5)
>>> c = a + b
>>> c.days
2
>>> c.seconds
37800
>>> c.seconds / 3600
10.5
>>> c.total_seconds() / 3600
58.5
>>>
```

If you need to represent specific dates and times, create `datetime` instances and use the standard mathematical operations to manipulate them. For example:

```
>>> from datetime import datetime
>>> a = datetime(2012, 9, 23)
>>> print(a + timedelta(days=10))
2012-10-03 00:00:00
>>>
>>> b = datetime(2012, 12, 21)
>>> d = b - a
>>> d.days
89
>>> now = datetime.today()
>>> print(now)
2012-12-21 14:54:43.094063
>>> print(now + timedelta(minutes=10))
2012-12-21 15:04:43.094063
>>>
```

When making calculations, it should be noted that `datetime` is aware of leap years. For example:

```

>>> a = datetime(2012, 3, 1)
>>> b = datetime(2012, 2, 28)
>>> a - b
datetime.timedelta(2)
>>> (a - b).days
2
>>> c = datetime(2013, 3, 1)
>>> d = datetime(2013, 2, 28)
>>> (c - d).days
1
>>>

```

## Discussion

For most basic date and time manipulation problems, the `datetime` module will suffice. If you need to perform more complex date manipulations, such as dealing with time zones, fuzzy time ranges, calculating the dates of holidays, and so forth, look at the [dateutil module](#).

To illustrate, many similar time calculations can be performed with the `dateutil.relativedelta` function. However, one notable feature is that it fills in some gaps pertaining to the handling of months (and their differing number of days). For instance:

```

>>> a = datetime(2012, 9, 23)
>>> a + timedelta(months=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'months' is an invalid keyword argument for this function
>>>

>>> from dateutil.relativedelta import relativedelta
>>> a + relativedelta(months=+1)
datetime.datetime(2012, 10, 23, 0, 0)
>>> a + relativedelta(months=+4)
datetime.datetime(2013, 1, 23, 0, 0)
>>>

>>> # Time between two dates
>>> b = datetime(2012, 12, 21)
>>> d = b - a
>>> d
datetime.timedelta(89)
>>> d = relativedelta(b, a)
>>> d
relativedelta(months=+2, days=+28)
>>> d.months
2
>>> d.days
28
>>>

```

## 3.13. Determining Last Friday's Date

### Problem

You want a general solution for finding a date for the last occurrence of a day of the week. Last Friday, for example.

### Solution

Python's `datetime` module has utility functions and classes to help perform calculations like this. A decent, generic solution to this problem looks like this:

```
from datetime import datetime, timedelta

weekdays = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
             'Friday', 'Saturday', 'Sunday']

def get_previous_byday(dayname, start_date=None):
    if start_date is None:
        start_date = datetime.today()
    day_num = start_date.weekday()
    day_num_target = weekdays.index(dayname)
    days_ago = (7 + day_num - day_num_target) % 7
    if days_ago == 0:
        days_ago = 7
    target_date = start_date - timedelta(days=days_ago)
    return target_date
```

Using this in an interpreter session would look like this:

```
>>> datetime.today() # For reference
datetime.datetime(2012, 8, 28, 22, 4, 30, 263076)
>>> get_previous_byday('Monday')
datetime.datetime(2012, 8, 27, 22, 3, 57, 29045)
>>> get_previous_byday('Tuesday') # Previous week, not today
datetime.datetime(2012, 8, 21, 22, 4, 12, 629771)
>>> get_previous_byday('Friday')
datetime.datetime(2012, 8, 24, 22, 5, 9, 911393)
>>>
```

The optional `start_date` can be supplied using another `datetime` instance. For example:

```
>>> get_previous_byday('Sunday', datetime(2012, 12, 21))
datetime.datetime(2012, 12, 16, 0, 0)
>>>
```

## Discussion

This recipe works by mapping the start date and the target date to their numeric position in the week (with Monday as day 0). Modular arithmetic is then used to figure out how many days ago the target date last occurred. From there, the desired date is calculated from the start date by subtracting an appropriate `timedelta` instance.

If you're performing a lot of date calculations like this, you may be better off installing the `python-dateutil` package instead. For example, here is an example of performing the same calculation using the `relativedelta()` function from `dateutil`:

```
>>> from datetime import datetime
>>> from dateutil.relativedelta import relativedelta
>>> from dateutil.rrule import *
>>> d = datetime.now()
>>> print(d)
2012-12-23 16:31:52.718111

>>> # Next Friday
>>> print(d + relativedelta(weekday=FR))
2012-12-28 16:31:52.718111
>>>

>>> # Last Friday
>>> print(d + relativedelta(weekday=FR(-1)))
2012-12-21 16:31:52.718111
>>>
```

## 3.14. Finding the Date Range for the Current Month

### Problem

You have some code that needs to loop over each date in the current month, and want an efficient way to calculate that date range.

### Solution

Looping over the dates doesn't require building a list of all the dates ahead of time. You can just calculate the starting and stopping date in the range, then use `datetime.timedelta` objects to increment the date as you go.

Here's a function that takes any `datetime` object, and returns a tuple containing the first date of the month and the starting date of the next month:

```
from datetime import datetime, date, timedelta
import calendar

def get_month_range(start_date=None):
    if start_date is None:
```

```

        start_date = date.today().replace(day=1)
        _, days_in_month = calendar.monthrange(start_date.year, start_date.month)
        end_date = start_date + timedelta(days=days_in_month)
        return (start_date, end_date)

```

With this in place, it's pretty simple to loop over the date range:

```

>>> a_day = timedelta(days=1)
>>> first_day, last_day = get_month_range()
>>> while first_day < last_day:
...     print(first_day)
...     first_day += a_day
...
2012-08-01
2012-08-02
2012-08-03
2012-08-04
2012-08-05
2012-08-06
2012-08-07
2012-08-08
2012-08-09
#... and so on...

```

## Discussion

This recipe works by first calculating a date corresponding to the first day of the month. A quick way to do this is to use the `replace()` method of a `date` or `datetime` object to simply set the `days` attribute to 1. One nice thing about the `replace()` method is that it creates the same kind of object that you started with. Thus, if the input was a `date` instance, the result is a `date`. Likewise, if the input was a `datetime` instance, you get a `datetime` instance.

After that, the `calendar.monthrange()` function is used to find out how many days are in the month in question. Any time you need to get basic information about calendars, the `calendar` module can be useful. `monthrange()` is only one such function that returns a tuple containing the day of the week along with the number of days in the month.

Once the number of days in the month is known, the ending date is calculated by adding an appropriate `timedelta` to the starting date. It's subtle, but an important aspect of this recipe is that the ending date is not to be included in the range (it is actually the first day of the next month). This mirrors the behavior of Python's slices and range operations, which also never include the end point.

To loop over the date range, standard math and comparison operators are used. For example, `timedelta` instances can be used to increment the date. The `<` operator is used to check whether a date comes before the ending date.

Ideally, it would be nice to create a function that works like the built-in `range()` function, but for dates. Fortunately, this is extremely easy to implement using a generator:

```
def date_range(start, stop, step):
    while start < stop:
        yield start
        start += step
```

Here is an example of it in use:

```
>>> for d in date_range(datetime(2012, 9, 1), datetime(2012,10,1),
                        timedelta(hours=6)):
...     print(d)
...
2012-09-01 00:00:00
2012-09-01 06:00:00
2012-09-01 12:00:00
2012-09-01 18:00:00
2012-09-02 00:00:00
2012-09-02 06:00:00
...
>>>
```

Again, a major part of the ease of implementation is that dates and times can be manipulated using standard math and comparison operators.

## 3.15. Converting Strings into Datetimes

### Problem

Your application receives temporal data in string format, but you want to convert those strings into `datetime` objects in order to perform nonstring operations on them.

### Solution

Python's standard `datetime` module is typically the easy solution for this. For example:

```
>>> from datetime import datetime
>>> text = '2012-09-20'
>>> y = datetime.strptime(text, '%Y-%m-%d')
>>> z = datetime.now()
>>> diff = z - y
>>> diff
datetime.timedelta(3, 77824, 177393)
>>>
```

### Discussion

The `datetime.strptime()` method supports a host of formatting codes, like `%Y` for the four-digit year and `%m` for the two-digit month. It's also worth noting that these format-

ting placeholders also work in reverse, in case you need to represent a `datetime` object in string output and make it look nice.

For example, let's say you have some code that generates a `datetime` object, but you need to format a nice, human-readable date to put in the header of an auto-generated letter or report:

```
>>> z
datetime.datetime(2012, 9, 23, 21, 37, 4, 177393)
>>> nice_z = datetime.strftime(z, '%A %B %d, %Y')
>>> nice_z
'Sunday September 23, 2012'
>>>
```

It's worth noting that the performance of `strftime()` is often much worse than you might expect, due to the fact that it's written in pure Python and it has to deal with all sorts of system locale settings. If you are parsing a lot of dates in your code and you know the precise format, you will probably get much better performance by cooking up a custom solution instead. For example, if you knew that the dates were of the form “YYYY-MM-DD,” you could write a function like this:

```
from datetime import datetime
def parse_ymd(s):
    year_s, mon_s, day_s = s.split('-')
    return datetime(int(year_s), int(mon_s), int(day_s))
```

When tested, this function runs over seven times faster than `datetime.strptime()`. This is probably something to consider if you're processing large amounts of data involving dates.

## 3.16. Manipulating Dates Involving Time Zones

### Problem

You had a conference call scheduled for December 21, 2012, at 9:30 a.m. in Chicago. At what local time did your friend in Bangalore, India, have to show up to attend?

### Solution

For almost any problem involving time zones, you should use the `pytz module`. This package provides the Olson time zone database, which is the de facto standard for time zone information found in many languages and operating systems.

A major use of `pytz` is in localizing simple dates created with the `datetime` library. For example, here is how you would represent a date in Chicago time:

```
>>> from datetime import datetime
>>> from pytz import timezone
```



```

>>> d = datetime(2012, 12, 21, 9, 30, 0)
>>> print(d)
2012-12-21 09:30:00
>>>

>>> # Localize the date for Chicago
>>> central = timezone('US/Central')
>>> loc_d = central.localize(d)
>>> print(loc_d)
2012-12-21 09:30:00-06:00
>>>

```

Once the date has been localized, it can be converted to other time zones. To find the same time in Bangalore, you would do this:

```

>>> # Convert to Bangalore time
>>> bang_d = loc_d.astimezone(timezone('Asia/Kolkata'))
>>> print(bang_d)
2012-12-21 21:00:00+05:30
>>>

```

If you are going to perform arithmetic with localized dates, you need to be particularly aware of daylight saving transitions and other details. For example, in 2013, U.S. standard daylight saving time started on March 13, 2:00 a.m. local time (at which point, time skipped ahead one hour). If you're performing naive arithmetic, you'll get it wrong. For example:

```

>>> d = datetime(2013, 3, 10, 1, 45)
>>> loc_d = central.localize(d)
>>> print(loc_d)
2013-03-10 01:45:00-06:00
>>> later = loc_d + timedelta(minutes=30)
>>> print(later)
2013-03-10 02:15:00-06:00      # WRONG! WRONG!
>>>

```

The answer is wrong because it doesn't account for the one-hour skip in the local time. To fix this, use the `normalize()` method of the time zone. For example:

```

>>> from datetime import timedelta
>>> later = central.normalize(loc_d + timedelta(minutes=30))
>>> print(later)
2013-03-10 03:15:00-05:00
>>>

```

## Discussion

To keep your head from completely exploding, a common strategy for localized date handling is to convert all dates to UTC time and to use that for all internal storage and manipulation. For example:

```
>>> print(loc_d)
2013-03-10 01:45:00-06:00
>>> utc_d = loc_d.astimezone(pytz.utc)
>>> print(utc_d)
2013-03-10 07:45:00+00:00
>>>
```

Once in UTC, you don't have to worry about issues related to daylight saving time and other matters. Thus, you can simply perform normal date arithmetic as before. Should you want to output the date in localized time, just convert it to the appropriate time zone afterward. For example:

```
>>> later_utc = utc_d + timedelta(minutes=30)
>>> print(later_utc.astimezone(central))
2013-03-10 03:15:00-05:00
>>>
```

One issue in working with time zones is simply figuring out what time zone names to use. For example, in this recipe, how was it known that “Asia/Kolkata” was the correct time zone name for India? To find out, you can consult the `pytz.country_timezones` dictionary using the ISO 3166 country code as a key. For example:

```
>>> pytz.country_timezones['IN']
['Asia/Kolkata']
>>>
```



By the time you read this, it's possible that the `pytz` module will be deprecated in favor of improved time zone support, as described in [PEP 431](#). Many of the same issues will still apply, however (e.g., advice using UTC dates, etc.).

---

# Iterators and Generators

Iteration is one of Python's strongest features. At a high level, you might simply view iteration as a way to process items in a sequence. However, there is so much more that is possible, such as creating your own iterator objects, applying useful iteration patterns in the `itertools` module, making generator functions, and so forth. This chapter aims to address common problems involving iteration.

## 4.1. Manually Consuming an Iterator

### Problem

You need to process items in an iterable, but for whatever reason, you can't or don't want to use a `for` loop.

### Solution

To manually consume an iterable, use the `next()` function and write your code to catch the `StopIteration` exception. For example, this example manually reads lines from a file:

```
with open('/etc/passwd') as f:
    try:
        while True:
            line = next(f)
            print(line, end='')
    except StopIteration:
        pass
```

Normally, `StopIteration` is used to signal the end of iteration. However, if you're using `next()` manually (as shown), you can also instruct it to return a terminating value, such as `None`, instead. For example:

```

with open('/etc/passwd') as f:
    while True:
        line = next(f, None)
        if line is None:
            break
        print(line, end='')

```

## Discussion

In most cases, the `for` statement is used to consume an iterable. However, every now and then, a problem calls for more precise control over the underlying iteration mechanism. Thus, it is useful to know what actually happens.

The following interactive example illustrates the basic mechanics of what happens during iteration:

```

>>> items = [1, 2, 3]
>>> # Get the iterator
>>> it = iter(items)      # Invokes items.__iter__()
>>> # Run the iterator
>>> next(it)              # Invokes it.__next__()
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>

```

Subsequent recipes in this chapter expand on iteration techniques, and knowledge of the basic iterator protocol is assumed. Be sure to tuck this first recipe away in your memory.

## 4.2. Delegating Iteration

### Problem

You have built a custom container object that internally holds a list, tuple, or some other iterable. You would like to make iteration work with your new container.

### Solution

Typically, all you need to do is define an `__iter__()` method that delegates iteration to the internally held container. For example:

```

class Node:
    def __init__(self, value):
        self._value = value
        self._children = []

    def __repr__(self):
        return 'Node({!r})'.format(self._value)

    def add_child(self, node):
        self._children.append(node)

    def __iter__(self):
        return iter(self._children)

# Example
if __name__ == '__main__':
    root = Node(0)
    child1 = Node(1)
    child2 = Node(2)
    root.add_child(child1)
    root.add_child(child2)
    for ch in root:
        print(ch)
    # Outputs Node(1), Node(2)

```

In this code, the `__iter__()` method simply forwards the iteration request to the internally held `_children` attribute.

## Discussion

Python's iterator protocol requires `__iter__()` to return a special iterator object that implements a `__next__()` method to carry out the actual iteration. If all you are doing is iterating over the contents of another container, you don't really need to worry about the underlying details of how it works. All you need to do is to forward the iteration request along.

The use of the `iter()` function here is a bit of a shortcut that cleans up the code. `iter(s)` simply returns the underlying iterator by calling `s.__iter__()`, much in the same way that `len(s)` invokes `s.__len__()`.

## 4.3. Creating New Iteration Patterns with Generators

### Problem

You want to implement a custom iteration pattern that's different than the usual built-in functions (e.g., `range()`, `reversed()`, etc.).

## Solution

If you want to implement a new kind of iteration pattern, define it using a generator function. Here's a generator that produces a range of floating-point numbers:

```
def frange(start, stop, increment):
    x = start
    while x < stop:
        yield x
        x += increment
```

To use such a function, you iterate over it using a for loop or use it with some other function that consumes an iterable (e.g., `sum()`, `list()`, etc.). For example:

```
>>> for n in frange(0, 4, 0.5):
...     print(n)
...
0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
>>> list(frange(0, 1, 0.125))
[0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875]
>>>
```

## Discussion

The mere presence of the `yield` statement in a function turns it into a generator. Unlike a normal function, a generator only runs in response to iteration. Here's an experiment you can try to see the underlying mechanics of how such a function works:

```
>>> def countdown(n):
...     print('Starting to count from', n)
...     while n > 0:
...         yield n
...         n -= 1
...     print('Done!')
...

>>> # Create the generator, notice no output appears
>>> c = countdown(3)
>>> c
<generator object countdown at 0x1006a0af0>

>>> # Run to first yield and emit a value
>>> next(c)
Starting to count from 3
3
```

```

>>> # Run to the next yield
>>> next(c)
2

>>> # Run to next yield
>>> next(c)
1

>>> # Run to next yield (iteration stops)
>>> next(c)
Done!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>

```

The key feature is that a generator function only runs in response to “next” operations carried out in iteration. Once a generator function returns, iteration stops. However, the `for` statement that’s usually used to iterate takes care of these details, so you don’t normally need to worry about them.

## 4.4. Implementing the Iterator Protocol

### Problem

You are building custom objects on which you would like to support iteration, but would like an easy way to implement the iterator protocol.

### Solution

By far, the easiest way to implement iteration on an object is to use a generator function. In [Recipe 4.2](#), a `Node` class was presented for representing tree structures. Perhaps you want to implement an iterator that traverses nodes in a depth-first pattern. Here is how you could do it:

```

class Node:
    def __init__(self, value):
        self._value = value
        self._children = []

    def __repr__(self):
        return 'Node({!r})'.format(self._value)

    def add_child(self, node):
        self._children.append(node)

    def __iter__(self):
        return iter(self._children)

```

```

def depth_first(self):
    yield self
    for c in self:
        yield from c.depth_first()

# Example
if __name__ == '__main__':
    root = Node(0)
    child1 = Node(1)
    child2 = Node(2)
    root.add_child(child1)
    root.add_child(child2)
    child1.add_child(Node(3))
    child1.add_child(Node(4))
    child2.add_child(Node(5))

    for ch in root.depth_first():
        print(ch)
    # Outputs Node(0), Node(1), Node(3), Node(4), Node(2), Node(5)

```

In this code, the `depth_first()` method is simple to read and describe. It first yields itself and then iterates over each child yielding the items produced by the child's `depth_first()` method (using `yield from`).

## Discussion

Python's iterator protocol requires `__iter__()` to return a special iterator object that implements a `__next__()` operation and uses a `StopIteration` exception to signal completion. However, implementing such objects can often be a messy affair. For example, the following code shows an alternative implementation of the `depth_first()` method using an associated iterator class:

```

class Node:
    def __init__(self, value):
        self._value = value
        self._children = []

    def __repr__(self):
        return 'Node({!r})'.format(self._value)

    def add_child(self, other_node):
        self._children.append(other_node)

    def __iter__(self):
        return iter(self._children)

    def depth_first(self):
        return DepthFirstIterator(self)

class DepthFirstIterator(object):

```



```

'''
Depth-first traversal
'''
def __init__(self, start_node):
    self._node = start_node
    self._children_iter = None
    self._child_iter = None

def __iter__(self):
    return self

def __next__(self):
    # Return myself if just started; create an iterator for children
    if self._children_iter is None:
        self._children_iter = iter(self._node)
        return self._node

    # If processing a child, return its next item
    elif self._child_iter:
        try:
            nextchild = next(self._child_iter)
            return nextchild
        except StopIteration:
            self._child_iter = None
            return next(self)

    # Advance to the next child and start its iteration
    else:
        self._child_iter = next(self._children_iter).depth_first()
        return next(self)

```

The `DepthFirstIterator` class works in the same way as the generator version, but it's a mess because the iterator has to maintain a lot of complex state about where it is in the iteration process. Frankly, nobody likes to write mind-bending code like that. Define your iterator as a generator and be done with it.

## 4.5. Iterating in Reverse

### Problem

You want to iterate in reverse over a sequence.

### Solution

Use the built-in `reversed()` function. For example:

```

>>> a = [1, 2, 3, 4]
>>> for x in reversed(a):
...     print(x)
...

```

4  
3  
2  
1

Reversed iteration only works if the object in question has a size that can be determined or if the object implements a `__reversed__()` special method. If neither of these can be satisfied, you'll have to convert the object into a list first. For example:

```
# Print a file backwards
f = open('somefile')
for line in reversed(list(f)):
    print(line, end='')
```

Be aware that turning an iterable into a list as shown could consume a lot of memory if it's large.

## Discussion

Many programmers don't realize that reversed iteration can be customized on user-defined classes if they implement the `__reversed__()` method. For example:

```
class Countdown:
    def __init__(self, start):
        self.start = start

    # Forward iterator
    def __iter__(self):
        n = self.start
        while n > 0:
            yield n
            n -= 1

    # Reverse iterator
    def __reversed__(self):
        n = 1
        while n <= self.start:
            yield n
            n += 1
```

Defining a reversed iterator makes the code much more efficient, as it's no longer necessary to pull the data into a list and iterate in reverse on the list.

## 4.6. Defining Generator Functions with Extra State

### Problem

You would like to define a generator function, but it involves extra state that you would like to expose to the user somehow.

## Solution

If you want a generator to expose extra state to the user, don't forget that you can easily implement it as a class, putting the generator function code in the `__iter__()` method. For example:

```
from collections import deque

class linehistory:
    def __init__(self, lines, histlen=3):
        self.lines = lines
        self.history = deque(maxlen=histlen)

    def __iter__(self):
        for lineno, line in enumerate(self.lines,1):
            self.history.append((lineno, line))
            yield line

    def clear(self):
        self.history.clear()
```

To use this class, you would treat it like a normal generator function. However, since it creates an instance, you can access internal attributes, such as the `history` attribute or the `clear()` method. For example:

```
with open('somefile.txt') as f:
    lines = linehistory(f)
    for line in lines:
        if 'python' in line:
            for lineno, hline in lines.history:
                print('{}:{}'.format(lineno, hline), end='')
```

## Discussion

With generators, it is easy to fall into a trap of trying to do everything with functions alone. This can lead to rather complicated code if the generator function needs to interact with other parts of your program in unusual ways (exposing attributes, allowing control via method calls, etc.). If this is the case, just use a class definition, as shown. Defining your generator in the `__iter__()` method doesn't change anything about how you write your algorithm. The fact that it's part of a class makes it easy for you to provide attributes and methods for users to interact with.

One potential subtlety with the method shown is that it might require an extra step of calling `iter()` if you are going to drive iteration using a technique other than a `for` loop. For example:

```
>>> f = open('somefile.txt')
>>> lines = linehistory(f)
>>> next(lines)
Traceback (most recent call last):
```

```

File "<stdin>", line 1, in <module>
TypeError: 'linehistory' object is not an iterator

>>> # Call iter() first, then start iterating
>>> it = iter(lines)
>>> next(it)
'hello world\n'
>>> next(it)
'this is a test\n'
>>>

```

## 4.7. Taking a Slice of an Iterator

### Problem

You want to take a slice of data produced by an iterator, but the normal slicing operator doesn't work.

### Solution

The `itertools.islice()` function is perfectly suited for taking slices of iterators and generators. For example:

```

>>> def count(n):
...     while True:
...         yield n
...         n += 1
...
>>> c = count(0)
>>> c[10:20]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'generator' object is not subscriptable

>>> # Now using islice()
>>> import itertools
>>> for x in itertools.islice(c, 10, 20):
...     print(x)
...
10
11
12
13
14
15
16
17
18
19
>>>

```

## Discussion

Iterators and generators can't normally be sliced, because no information is known about their length (and they don't implement indexing). The result of `islice()` is an iterator that produces the desired slice items, but it does this by consuming and discarding all of the items up to the starting slice index. Further items are then produced by the `islice` object until the ending index has been reached.

It's important to emphasize that `islice()` will consume data on the supplied iterator. Since iterators can't be rewound, that is something to consider. If it's important to go back, you should probably just turn the data into a list first.

## 4.8. Skipping the First Part of an Iterable

### Problem

You want to iterate over items in an iterable, but the first few items aren't of interest and you just want to discard them.

### Solution

The `itertools` module has a few functions that can be used to address this task. The first is the `itertools.dropwhile()` function. To use it, you supply a function and an iterable. The returned iterator discards the first items in the sequence as long as the supplied function returns `True`. Afterward, the entirety of the sequence is produced.

To illustrate, suppose you are reading a file that starts with a series of comment lines. For example:

```
>>> with open('/etc/passwd') as f:
...     for line in f:
...         print(line, end='')
...
##
# User Database
#
# Note that this file is consulted directly only when the system is running
# in single-user mode. At other times, this information is provided by
# Open Directory.
...
##
nobody:*:~:~:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
...
>>>
```

If you want to skip all of the initial comment lines, here's one way to do it:

```

>>> from itertools import dropwhile
>>> with open('/etc/passwd') as f:
...     for line in dropwhile(lambda line: line.startswith('#'), f):
...         print(line, end='')
...
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
...
>>>

```

This example is based on skipping the first items according to a test function. If you happen to know the exact number of items you want to skip, then you can use `iter tools.islice()` instead. For example:

```

>>> from itertools import islice
>>> items = ['a', 'b', 'c', 1, 4, 10, 15]
>>> for x in islice(items, 3, None):
...     print(x)
...
1
4
10
15
>>>

```

In this example, the last `None` argument to `islice()` is required to indicate that you want everything *beyond* the first three items as opposed to only the first three items (e.g., a slice of `[3:]` as opposed to a slice of `[:3]`).

## Discussion

The `dropwhile()` and `islice()` functions are mainly convenience functions that you can use to avoid writing rather messy code such as this:

```

with open('/etc/passwd') as f:
    # Skip over initial comments
    while True:
        line = next(f, '')
        if not line.startswith('#'):
            break

    # Process remaining lines
    while line:
        # Replace with useful processing
        print(line, end='')
        line = next(f, None)

```

Discarding the first part of an iterable is also slightly different than simply filtering all of it. For example, the first part of this recipe might be rewritten as follows:

```

with open('/etc/passwd') as f:
    lines = (line for line in f if not line.startswith('#'))

```

```
for line in lines:
    print(line, end='')
```

This will obviously discard the comment lines at the start, but will also discard all such lines throughout the entire file. On the other hand, the solution only discards items until an item no longer satisfies the supplied test. After that, all subsequent items are returned with no filtering.

Last, but not least, it should be emphasized that this recipe works with all iterables, including those whose size can't be determined in advance. This includes generators, files, and similar kinds of objects.

## 4.9. Iterating Over All Possible Combinations or Permutations

### Problem

You want to iterate over all of the possible combinations or permutations of a collection of items.

### Solution

The `itertools` module provides three functions for this task. The first of these—`itertools.permutations()`—takes a collection of items and produces a sequence of tuples that rearranges all of the items into all possible permutations (i.e., it shuffles them into all possible configurations). For example:

```
>>> items = ['a', 'b', 'c']
>>> from itertools import permutations
>>> for p in permutations(items):
...     print(p)
...
('a', 'b', 'c')
('a', 'c', 'b')
('b', 'a', 'c')
('b', 'c', 'a')
('c', 'a', 'b')
('c', 'b', 'a')
>>>
```

If you want all permutations of a smaller length, you can give an optional `length` argument. For example:

```
>>> for p in permutations(items, 2):
...     print(p)
...
('a', 'b')
('a', 'c')
```

```

('b', 'a')
('b', 'c')
('c', 'a')
('c', 'b')
>>>

```

Use `itertools.combinations()` to produce a sequence of combinations of items taken from the input. For example:

```

>>> from itertools import combinations
>>> for c in combinations(items, 3):
...     print(c)
...
('a', 'b', 'c')
>>> for c in combinations(items, 2):
...     print(c)
...
('a', 'b')
('a', 'c')
('b', 'c')
>>> for c in combinations(items, 1):
...     print(c)
...
('a',)
('b',)
('c',)
>>>

```

For `combinations()`, the actual order of the elements is not considered. That is, the combination `('a', 'b')` is considered to be the same as `('b', 'a')` (which is not produced).

When producing combinations, chosen items are removed from the collection of possible candidates (i.e., if `'a'` has already been chosen, then it is removed from consideration). The `itertools.combinations_with_replacement()` function relaxes this, and allows the same item to be chosen more than once. For example:

```

>>> for c in combinations_with_replacement(items, 3):
...     print(c)
...
('a', 'a', 'a')
('a', 'a', 'b')
('a', 'a', 'c')
('a', 'b', 'b')
('a', 'b', 'c')
('a', 'c', 'c')
('b', 'b', 'b')
('b', 'b', 'c')
('b', 'c', 'c')
('c', 'c', 'c')
>>>

```



## Discussion

This recipe demonstrates only some of the power found in the `itertools` module. Although you could certainly write code to produce permutations and combinations yourself, doing so would probably require more than a fair bit of thought. When faced with seemingly complicated iteration problems, it always pays to look at `itertools` first. If the problem is common, chances are a solution is already available.

## 4.10. Iterating Over the Index-Value Pairs of a Sequence

### Problem

You want to iterate over a sequence, but would like to keep track of which element of the sequence is currently being processed.

### Solution

The built-in `enumerate()` function handles this quite nicely:

```
>>> my_list = ['a', 'b', 'c']
>>> for idx, val in enumerate(my_list):
...     print(idx, val)
...
0 a
1 b
2 c
```

For printing output with canonical line numbers (where you typically start the numbering at 1 instead of 0), you can pass in a `start` argument:

```
>>> my_list = ['a', 'b', 'c']
>>> for idx, val in enumerate(my_list, 1):
...     print(idx, val)
...
1 a
2 b
3 c
```

This case is especially useful for tracking line numbers in files should you want to use a line number in an error message:

```
def parse_data(filename):
    with open(filename, 'rt') as f:
        for lineno, line in enumerate(f, 1):
            fields = line.split()
            try:
                count = int(fields[1])
            ...
        except ValueError as e:
            print('Line {}: Parse error: {}'.format(lineno, e))
```

`enumerate()` can be handy for keeping track of the offset into a list for occurrences of certain values, for example. So, if you want to map words in a file to the lines in which they occur, it can easily be accomplished using `enumerate()` to map each word to the line offset in the file where it was found:

```
word_summary = defaultdict(list)

with open('myfile.txt', 'r') as f:
    lines = f.readlines()

for idx, line in enumerate(lines):
    # Create a list of words in current line
    words = [w.strip().lower() for w in line.split()]
    for word in words:
        word_summary[word].append(idx)
```

If you print `word_summary` after processing the file, it'll be a dictionary (a default dict to be precise), and it'll have a key for each word. The value for each word-key will be a list of line numbers that word occurred on. If the word occurred twice on a single line, that line number will be listed twice, making it possible to identify various simple metrics about the text.

## Discussion

`enumerate()` is a nice shortcut for situations where you might be inclined to keep your own counter variable. You could write code like this:

```
lineno = 1
for line in f:
    # Process line
    ...
    lineno += 1
```

But it's usually much more elegant (and less error prone) to use `enumerate()` instead:

```
for lineno, line in enumerate(f):
    # Process line
    ...
```

The value returned by `enumerate()` is an instance of an `enumerate` object, which is an iterator that returns successive tuples consisting of a counter and the value returned by calling `next()` on the sequence you've passed in.

Although a minor point, it's worth mentioning that sometimes it is easy to get tripped up when applying `enumerate()` to a sequence of tuples that are also being unpacked. To do it, you have to write code like this:

```
data = [ (1, 2), (3, 4), (5, 6), (7, 8) ]

# Correct!
for n, (x, y) in enumerate(data):
```

```

...
# Error!
for n, x, y in enumerate(data):
...

```

## 4.11. Iterating Over Multiple Sequences Simultaneously

### Problem

You want to iterate over the items contained in more than one sequence at a time.

### Solution

To iterate over more than one sequence simultaneously, use the `zip()` function. For example:

```

>>> xpts = [1, 5, 4, 2, 10, 7]
>>> ypts = [101, 78, 37, 15, 62, 99]
>>> for x, y in zip(xpts, ypts):
...     print(x,y)
...
1 101
5 78
4 37
2 15
10 62
7 99
>>>

```

`zip(a, b)` works by creating an iterator that produces tuples `(x, y)` where `x` is taken from `a` and `y` is taken from `b`. Iteration stops whenever one of the input sequences is exhausted. Thus, the length of the iteration is the same as the length of the shortest input. For example:

```

>>> a = [1, 2, 3]
>>> b = ['w', 'x', 'y', 'z']
>>> for i in zip(a,b):
...     print(i)
...
(1, 'w')
(2, 'x')
(3, 'y')
>>>

```

If this behavior is not desired, use `itertools.zip_longest()` instead. For example:

```

>>> from itertools import zip_longest
>>> for i in zip_longest(a,b):
...     print(i)
...

```

```

(1, 'w')
(2, 'x')
(3, 'y')
(None, 'z')
>>> for i in zip_longest(a, b, fillvalue=0):
...     print(i)
...
(1, 'w')
(2, 'x')
(3, 'y')
(0, 'z')
>>>

```

## Discussion

`zip()` is commonly used whenever you need to pair data together. For example, suppose you have a list of column headers and column values like this:

```

headers = ['name', 'shares', 'price']
values = ['ACME', 100, 490.1]

```

Using `zip()`, you can pair the values together to make a dictionary like this:

```

s = dict(zip(headers, values))

```

Alternatively, if you are trying to produce output, you can write code like this:

```

for name, val in zip(headers, values):
    print(name, '=', val)

```

It's less common, but `zip()` can be passed more than two sequences as input. For this case, the resulting tuples have the same number of items in them as the number of input sequences. For example:

```

>>> a = [1, 2, 3]
>>> b = [10, 11, 12]
>>> c = ['x', 'y', 'z']
>>> for i in zip(a, b, c):
...     print(i)
...
(1, 10, 'x')
(2, 11, 'y')
(3, 12, 'z')
>>>

```

Last, but not least, it's important to emphasize that `zip()` creates an iterator as a result. If you need the paired values stored in a list, use the `list()` function. For example:

```

>>> zip(a, b)
<zip object at 0x1007001b8>
>>> list(zip(a, b))
[(1, 10), (2, 11), (3, 12)]
>>>

```

## 4.12. Iterating on Items in Separate Containers

### Problem

You need to perform the same operation on many objects, but the objects are contained in different containers, and you'd like to avoid nested loops without losing the readability of your code.

### Solution

The `itertools.chain()` method can be used to simplify this task. It takes a list of iterables as input, and returns an iterator that effectively masks the fact that you're really acting on multiple containers. To illustrate, consider this example:

```
>>> from itertools import chain
>>> a = [1, 2, 3, 4]
>>> b = ['x', 'y', 'z']
>>> for x in chain(a, b):
...     print(x)
...
1
2
3
4
x
y
z
>>>
```

A common use of `chain()` is in programs where you would like to perform certain operations on all of the items at once but the items are pooled into different working sets. For example:

```
# Various working sets of items
active_items = set()
inactive_items = set()

# Iterate over all items
for item in chain(active_items, inactive_items):
    # Process item
    ...
```

This solution is much more elegant than using two separate loops, as in the following:

```
for item in active_items:
    # Process item
    ...

for item in inactive_items:
    # Process item
    ...
```

## Discussion

`itertools.chain()` accepts one or more iterables as arguments. It then works by creating an iterator that successively consumes and returns the items produced by each of the supplied iterables you provided. It's a subtle distinction, but `chain()` is more efficient than first combining the sequences and iterating. For example:

```
# Inefficient
for x in a + b:
    ...

# Better
for x in chain(a, b):
    ...
```

In the first case, the operation `a + b` creates an entirely new sequence and additionally requires `a` and `b` to be of the same type. `chain()` performs no such operation, so it's far more efficient with memory if the input sequences are large and it can be easily applied when the iterables in question are of different types.

## 4.13. Creating Data Processing Pipelines

### Problem

You want to process data iteratively in the style of a data processing pipeline (similar to Unix pipes). For instance, you have a huge amount of data that needs to be processed, but it can't fit entirely into memory.

### Solution

Generator functions are a good way to implement processing pipelines. To illustrate, suppose you have a huge directory of log files that you want to process:

```
foo/
  access-log-012007.gz
  access-log-022007.gz
  access-log-032007.gz
  ...
  access-log-012008
bar/
  access-log-092007.bz2
  ...
  access-log-022008
```

Suppose each file contains lines of data like this:

```
124.115.6.12 - - [10/Jul/2012:00:18:50 -0500] "GET /robots.txt ..." 200 71
210.212.209.67 - - [10/Jul/2012:00:18:51 -0500] "GET /ply/ ..." 200 11875
210.212.209.67 - - [10/Jul/2012:00:18:51 -0500] "GET /favicon.ico ..." 404 369
```

```
61.135.216.105 - - [10/Jul/2012:00:20:04 -0500] "GET /blog/atom.xml ..." 304 -  
...
```

To process these files, you could define a collection of small generator functions that perform specific self-contained tasks. For example:

```
import os  
import fnmatch  
import gzip  
import bz2  
import re  
  
def gen_find(filepat, top):  
    """  
    Find all filenames in a directory tree that match a shell wildcard pattern  
    """  
    for path, dirlist, filelist in os.walk(top):  
        for name in fnmatch.filter(filelist, filepat):  
            yield os.path.join(path, name)  
  
def gen_opener(filenames):  
    """  
    Open a sequence of filenames one at a time producing a file object.  
    The file is closed immediately when proceeding to the next iteration.  
    """  
    for filename in filenames:  
        if filename.endswith('.gz'):  
            f = gzip.open(filename, 'rt')  
        elif filename.endswith('.bz2'):  
            f = bz2.open(filename, 'rt')  
        else:  
            f = open(filename, 'rt')  
        yield f  
        f.close()  
  
def gen_concatenate(iterators):  
    """  
    Chain a sequence of iterators together into a single sequence.  
    """  
    for it in iterators:  
        yield from it  
  
def gen_grep(pattern, lines):  
    """  
    Look for a regex pattern in a sequence of lines  
    """  
    pat = re.compile(pattern)  
    for line in lines:  
        if pat.search(line):  
            yield line
```

You can now easily stack these functions together to make a processing pipeline. For example, to find all log lines that contain the word *python*, you would just do this:

```

lognames = gen_find('access-log*', 'www')
files = gen_opener(lognames)
lines = gen_concatenate(files)
pylines = gen_grep('(?i)python', lines)
for line in pylines:
    print(line)

```

If you want to extend the pipeline further, you can even feed the data in generator expressions. For example, this version finds the number of bytes transferred and sums the total:

```

lognames = gen_find('access-log*', 'www')
files = gen_opener(lognames)
lines = gen_concatenate(files)
pylines = gen_grep('(?i)python', lines)
bytecolumn = (line.rsplit(None,1)[1] for line in pylines)
bytes = (int(x) for x in bytecolumn if x != '-')
print('Total', sum(bytes))

```

## Discussion

Processing data in a pipelined manner works well for a wide variety of other problems, including parsing, reading from real-time data sources, periodic polling, and so on.

In understanding the code, it is important to grasp that the `yield` statement acts as a kind of data producer whereas a `for` loop acts as a data consumer. When the generators are stacked together, each `yield` feeds a single item of data to the next stage of the pipeline that is consuming it with iteration. In the last example, the `sum()` function is actually driving the entire program, pulling one item at a time out of the pipeline of generators.

One nice feature of this approach is that each generator function tends to be small and self-contained. As such, they are easy to write and maintain. In many cases, they are so general purpose that they can be reused in other contexts. The resulting code that glues the components together also tends to read like a simple recipe that is easily understood.

The memory efficiency of this approach can also not be overstated. The code shown would still work even if used on a massive directory of files. In fact, due to the iterative nature of the processing, very little memory would be used at all.

There is a bit of extreme subtlety involving the `gen_concatenate()` function. The purpose of this function is to concatenate input sequences together into one long sequence of lines. The `itertools.chain()` function performs a similar function, but requires that all of the chained iterables be specified as arguments. In the case of this particular recipe, doing that would involve a statement such as `lines = itertools.chain(*files)`, which would cause the `gen_opener()` generator to be fully consumed. Since that generator is producing a sequence of open files that are immediately



closed in the next iteration step, `chain()` can't be used. The solution shown avoids this issue.

Also appearing in the `gen_concatenate()` function is the use of `yield from` to delegate to a subgenerator. The statement `yield from` it simply makes `gen_concatenate()` emit all of the values produced by the generator it. This is described further in [Recipe 4.14](#).

Last, but not least, it should be noted that a pipelined approach doesn't always work for every data handling problem. Sometimes you just need to work with all of the data at once. However, even in that case, using generator pipelines can be a way to logically break a problem down into a kind of workflow.

David Beazley has written extensively about these techniques in his “[Generator Tricks for Systems Programmers](#)” tutorial presentation. Consult that for even more examples.

## 4.14. Flattening a Nested Sequence

### Problem

You have a nested sequence that you want to flatten into a single list of values.

### Solution

This is easily solved by writing a recursive generator function involving a `yield from` statement. For example:

```
from collections import Iterable

def flatten(items, ignore_types=(str, bytes)):
    for x in items:
        if isinstance(x, Iterable) and not isinstance(x, ignore_types):
            yield from flatten(x)
        else:
            yield x

items = [1, 2, [3, 4, [5, 6], 7], 8]

# Produces 1 2 3 4 5 6 7 8
for x in flatten(items):
    print(x)
```

In the code, the `isinstance(x, Iterable)` simply checks to see if an item is iterable. If so, `yield from` is used to emit all of its values as a kind of subroutine. The end result is a single sequence of output with no nesting.

The extra argument `ignore_types` and the check for `not isinstance(x, ignore_types)` is there to prevent strings and bytes from being interpreted as iterables

and expanded as individual characters. This allows nested lists of strings to work in the way that most people would expect. For example:

```
>>> items = ['Dave', 'Paula', ['Thomas', 'Lewis']]
>>> for x in flatten(items):
...     print(x)
...
Dave
Paula
Thomas
Lewis
>>>
```

## Discussion

The `yield from` statement is a nice shortcut to use if you ever want to write generators that call other generators as subroutines. If you don't use it, you need to write code that uses an extra `for` loop. For example:

```
def flatten(items, ignore_types=(str, bytes)):
    for x in items:
        if isinstance(x, Iterable) and not isinstance(x, ignore_types):
            for i in flatten(x):
                yield i
        else:
            yield x
```

Although it's only a minor change, the `yield from` statement just feels better and leads to cleaner code.

As noted, the extra check for strings and bytes is there to prevent the expansion of those types into individual characters. If there are other types that you don't want expanded, you can supply a different value for the `ignore_types` argument.

Finally, it should be noted that `yield from` has a more important role in advanced programs involving coroutines and generator-based concurrency. See [Recipe 12.12](#) for another example.

## 4.15. Iterating in Sorted Order Over Merged Sorted Iterables

### Problem

You have a collection of sorted sequences and you want to iterate over a sorted sequence of them all merged together.

## Solution

The `heapq.merge()` function does exactly what you want. For example:

```
>>> import heapq
>>> a = [1, 4, 7, 10]
>>> b = [2, 5, 6, 11]
>>> for c in heapq.merge(a, b):
...     print(c)
...
1
2
4
5
6
7
10
11
```

## Discussion

The iterative nature of `heapq.merge` means that it never reads any of the supplied sequences all at once. This means that you can use it on very long sequences with very little overhead. For instance, here is an example of how you would merge two sorted files:

```
import heapq

with open('sorted_file_1', 'rt') as file1, \
     open('sorted_file_2') 'rt' as file2, \
     open('merged_file', 'wt') as outf:

    for line in heapq.merge(file1, file2):
        outf.write(line)
```

It's important to emphasize that `heapq.merge()` requires that all of the input sequences already be sorted. In particular, it does not first read all of the data into a heap or do any preliminary sorting. Nor does it perform any kind of validation of the inputs to check if they meet the ordering requirements. Instead, it simply examines the set of items from the front of each input sequence and emits the smallest one found. A new item from the chosen sequence is then read, and the process repeats itself until all input sequences have been fully consumed.

## 4.16. Replacing Infinite while Loops with an Iterator

### Problem

You have code that uses a `while` loop to iteratively process data because it involves a function or some kind of unusual test condition that doesn't fall into the usual iteration pattern.

### Solution

A somewhat common scenario in programs involving I/O is to write code like this:

```
CHUNKSIZE = 8192

def reader(s):
    while True:
        data = s.recv(CHUNKSIZE)
        if data == b'':
            break
        process_data(data)
```

Such code can often be replaced using `iter()`, as follows:

```
def reader(s):
    for chunk in iter(lambda: s.recv(CHUNKSIZE), b''):
        process_data(chunk)
```

If you're a bit skeptical that it might work, you can try a similar example involving files. For example:

```
>>> import sys
>>> f = open('/etc/passwd')
>>> for chunk in iter(lambda: f.read(10), ''):
...     n = sys.stdout.write(chunk)
...
nobody:*:-2:-2:Unprivileged User:/var/empty:/usr/bin/false
root:*:0:0:System Administrator:/var/root:/bin/sh
daemon:*:1:1:System Services:/var/root:/usr/bin/false
_uucp:*:4:4:Unix to Unix Copy Protocol:/var/spool/uucp:/usr/sbin/uucico
...
>>>
```

### Discussion

A little-known feature of the built-in `iter()` function is that it optionally accepts a zero-argument callable and sentinel (terminating) value as inputs. When used in this way, it creates an iterator that repeatedly calls the supplied callable over and over again until it returns the value given as a sentinel.

This particular approach works well with certain kinds of repeatedly called functions, such as those involving I/O. For example, if you want to read data in chunks from sockets or files, you usually have to repeatedly execute `read()` or `recv()` calls followed by an end-of-file test. This recipe simply takes these two features and combines them together into a single `iter()` call. The use of `lambda` in the solution is needed to create a callable that takes no arguments, yet still supplies the desired size argument to `recv()` or `read()`.



All programs need to perform input and output. This chapter covers common idioms for working with different kinds of files, including text and binary files, file encodings, and other related matters. Techniques for manipulating filenames and directories are also covered.

## 5.1. Reading and Writing Text Data

### Problem

You need to read or write text data, possibly in different text encodings such as ASCII, UTF-8, or UTF-16.

### Solution

Use the `open()` function with mode `rt` to read a text file. For example:

```
# Read the entire file as a single string
with open('somefile.txt', 'rt') as f:
    data = f.read()

# Iterate over the lines of the file
with open('somefile.txt', 'rt') as f:
    for line in f:
        # process line
    ...
```

Similarly, to write a text file, use `open()` with mode `wt` to write a file, clearing and overwriting the previous contents (if any). For example:

```
# Write chunks of text data
with open('somefile.txt', 'wt') as f:
    f.write(text1)
```

```

f.write(text2)
...

# Redirected print statement
with open('somefile.txt', 'wt') as f:
    print(line1, file=f)
    print(line2, file=f)
...

```

To append to the end of an existing file, use `open()` with mode `at`.

By default, files are read/written using the system default text encoding, as can be found in `sys.getdefaultencoding()`. On most machines, this is set to `utf-8`. If you know that the text you are reading or writing is in a different encoding, supply the optional encoding parameter to `open()`. For example:

```

with open('somefile.txt', 'rt', encoding='latin-1') as f:
    ...

```

Python understands several hundred possible text encodings. However, some of the more common encodings are `ascii`, `latin-1`, `utf-8`, and `utf-16`. UTF-8 is usually a safe bet if working with web applications. `ascii` corresponds to the 7-bit characters in the range U+0000 to U+007F. `latin-1` is a direct mapping of bytes 0-255 to Unicode characters U+0000 to U+00FF. `latin-1` encoding is notable in that it will never produce a decoding error when reading text of a possibly unknown encoding. Reading a file as `latin-1` might not produce a completely correct text decoding, but it still might be enough to extract useful data out of it. Also, if you later write the data back out, the original input data will be preserved.

## Discussion

Reading and writing text files is typically very straightforward. However, there are a number of subtle aspects to keep in mind. First, the use of the `with` statement in the examples establishes a context in which the file will be used. When control leaves the `with` block, the file will be closed automatically. You don't need to use the `with` statement, but if you don't use it, make sure you remember to close the file:

```

f = open('somefile.txt', 'rt')
data = f.read()
f.close()

```

Another minor complication concerns the recognition of newlines, which are different on Unix and Windows (i.e., `\n` versus `\r\n`). By default, Python operates in what's known as "universal newline" mode. In this mode, all common newline conventions are recognized, and newline characters are converted to a single `\n` character while reading. Similarly, the newline character `\n` is converted to the system default newline character



on output. If you don't want this translation, supply the `newline=''` argument to `open()`, like this:

```
# Read with disabled newline translation
with open('somefile.txt', 'rt', newline='') as f:
    ...
```

To illustrate the difference, here's what you will see on a Unix machine if you read the contents of a Windows-encoded text file containing the raw data `hello world!\r\n`:

```
>>> # Newline translation enabled (the default)
>>> f = open('hello.txt', 'rt')
>>> f.read()
'hello world!\n'

>>> # Newline translation disabled
>>> g = open('hello.txt', 'rt', newline='')
>>> g.read()
'hello world!\r\n'
>>>
```

A final issue concerns possible encoding errors in text files. When reading or writing a text file, you might encounter an encoding or decoding error. For instance:

```
>>> f = open('sample.txt', 'rt', encoding='ascii')
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.3/encodings/ascii.py", line 26, in decode
    return codecs.ascii_decode(input, self.errors)[0]
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in position
12: ordinal not in range(128)
>>>
```

If you get this error, it usually means that you're not reading the file in the correct encoding. You should carefully read the specification of whatever it is that you're reading and check that you're doing it right (e.g., reading data as UTF-8 instead of Latin-1 or whatever it needs to be). If encoding errors are still a possibility, you can supply an optional `errors` argument to `open()` to deal with the errors. Here are a few samples of common error handling schemes:

```
>>> # Replace bad chars with Unicode U+fffd replacement char
>>> f = open('sample.txt', 'rt', encoding='ascii', errors='replace')
>>> f.read()
'Spicy Jalape?o!'

>>> # Ignore bad chars entirely
>>> g = open('sample.txt', 'rt', encoding='ascii', errors='ignore')
>>> g.read()
'Spicy Jalapeo!'
>>>
```

If you're constantly fiddling with the `encoding` and `errors` arguments to `open()` and doing lots of hacks, you're probably making life more difficult than it needs to be. The number one rule with text is that you simply need to make sure you're always using the proper text encoding. When in doubt, use the default setting (typically UTF-8).

## 5.2. Printing to a File

### Problem

You want to redirect the output of the `print()` function to a file.

### Solution

Use the `file` keyword argument to `print()`, like this:

```
with open('somefile.txt', 'rt') as f:
    print('Hello World!', file=f)
```

### Discussion

There's not much more to printing to a file other than this. However, make sure that the file is opened in text mode. Printing will fail if the underlying file is in binary mode.

## 5.3. Printing with a Different Separator or Line Ending

### Problem

You want to output data using `print()`, but you also want to change the separator character or line ending.

### Solution

Use the `sep` and `end` keyword arguments to `print()` to change the output as you wish. For example:

```
>>> print('ACME', 50, 91.5)
ACME 50 91.5
>>> print('ACME', 50, 91.5, sep=',')
ACME,50,91.5
>>> print('ACME', 50, 91.5, sep=',', end='!!\n')
ACME,50,91.5!!
>>>
```

Use of the `end` argument is also how you suppress the output of newlines in output. For example:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
>>> for i in range(5):
...     print(i, end=' ')
...
0 1 2 3 4 >>>
```

## Discussion

Using `print()` with a different item separator is often the easiest way to output data when you need something other than a space separating the items. Sometimes you'll see programmers using `str.join()` to accomplish the same thing. For example:

```
>>> print(', '.join('ACME', '50', '91.5'))
ACME,50,91.5
>>>
```

The problem with `str.join()` is that it only works with strings. This means that it's often necessary to perform various acrobatics to get it to work. For example:

```
>>> row = ('ACME', 50, 91.5)
>>> print(', '.join(row))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sequence item 1: expected str instance, int found
>>> print(', '.join(str(x) for x in row))
ACME,50,91.5
>>>
```

Instead of doing that, you could just write the following:

```
>>> print(*row, sep=', ')
ACME,50,91.5
>>>
```

## 5.4. Reading and Writing Binary Data

### Problem

You need to read or write binary data, such as that found in images, sound files, and so on.

## Solution

Use the `open()` function with mode `rb` or `wb` to read or write binary data. For example:

```
# Read the entire file as a single byte string
with open('somefile.bin', 'rb') as f:
    data = f.read()

# Write binary data to a file
with open('somefile.bin', 'wb') as f:
    f.write(b'Hello World')
```

When reading binary, it is important to stress that all data returned will be in the form of byte strings, not text strings. Similarly, when writing, you must supply data in the form of objects that expose data as bytes (e.g., byte strings, `bytearray` objects, etc.).

## Discussion

When reading binary data, the subtle semantic differences between byte strings and text strings pose a potential gotcha. In particular, be aware that indexing and iteration return integer byte values instead of byte strings. For example:

```
>>> # Text string
>>> t = 'Hello World'
>>> t[0]
'H'
>>> for c in t:
...     print(c)
...
H
e
l
l
o
...
>>> # Byte string
>>> b = b'Hello World'
>>> b[0]
72
>>> for c in b:
...     print(c)
...
72
101
108
108
111
...
>>>
```

If you ever need to read or write text from a binary-mode file, make sure you remember to decode or encode it. For example:

```
with open('somefile.bin', 'rb') as f:
    data = f.read(16)
    text = data.decode('utf-8')

with open('somefile.bin', 'wb') as f:
    text = 'Hello World'
    f.write(text.encode('utf-8'))
```

A lesser-known aspect of binary I/O is that objects such as arrays and C structures can be used for writing without any kind of intermediate conversion to a bytes object. For example:

```
import array
nums = array.array('i', [1, 2, 3, 4])
with open('data.bin', 'wb') as f:
    f.write(nums)
```

This applies to any object that implements the so-called “buffer interface,” which directly exposes an underlying memory buffer to operations that can work with it. Writing binary data is one such operation.

Many objects also allow binary data to be directly read into their underlying memory using the `readinto()` method of files. For example:

```
>>> import array
>>> a = array.array('i', [0, 0, 0, 0, 0, 0, 0, 0])
>>> with open('data.bin', 'rb') as f:
...     f.readinto(a)
...
16
>>> a
array('i', [1, 2, 3, 4, 0, 0, 0, 0])
>>>
```

However, great care should be taken when using this technique, as it is often platform specific and may depend on such things as the word size and byte ordering (i.e., big endian versus little endian). See [Recipe 5.9](#) for another example of reading binary data into a mutable buffer.

## 5.5. Writing to a File That Doesn’t Already Exist

### Problem

You want to write data to a file, but only if it doesn’t already exist on the filesystem.

## Solution

This problem is easily solved by using the little-known `x` mode to `open()` instead of the usual `w` mode. For example:

```
>>> with open('somefile', 'wt') as f:
...     f.write('Hello\n')
...
>>> with open('somefile', 'xt') as f:
...     f.write('Hello\n')
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileExistsError: [Errno 17] File exists: 'somefile'
>>>
```

If the file is binary mode, use mode `xb` instead of `xt`.

## Discussion

This recipe illustrates an extremely elegant solution to a problem that sometimes arises when writing files (i.e., accidentally overwriting an existing file). An alternative solution is to first test for the file like this:

```
>>> import os
>>> if not os.path.exists('somefile'):
...     with open('somefile', 'wt') as f:
...         f.write('Hello\n')
... else:
...     print('File already exists!')
...
File already exists!
>>>
```

Clearly, using the `x` file mode is a lot more straightforward. It is important to note that the `x` mode is a Python 3 specific extension to the `open()` function. In particular, no such mode exists in earlier Python versions or the underlying C libraries used in Python's implementation.

## 5.6. Performing I/O Operations on a String

### Problem

You want to feed a text or binary string to code that's been written to operate on file-like objects instead.

## Solution

Use the `io.StringIO()` and `io.BytesIO()` classes to create file-like objects that operate on string data. For example:

```
>>> s = io.StringIO()
>>> s.write('Hello World\n')
12
>>> print('This is a test', file=s)
15
>>> # Get all of the data written so far
>>> s.getvalue()
'Hello World\nThis is a test\n'
>>>

>>> # Wrap a file interface around an existing string
>>> s = io.StringIO('Hello\nWorld\n')
>>> s.read(4)
'Hell'
>>> s.read()
'o\nWorld\n'
>>>
```

The `io.StringIO` class should only be used for text. If you are operating with binary data, use the `io.BytesIO` class instead. For example:

```
>>> s = io.BytesIO()
>>> s.write(b'binary data')
>>> s.getvalue()
b'binary data'
>>>
```

## Discussion

The `StringIO` and `BytesIO` classes are most useful in scenarios where you need to mimic a normal file for some reason. For example, in unit tests, you might use `StringIO` to create a file-like object containing test data that's fed into a function that would otherwise work with a normal file.

Be aware that `StringIO` and `BytesIO` instances don't have a proper integer file-descriptor. Thus, they do not work with code that requires the use of a real system-level file such as a file, pipe, or socket.

## 5.7. Reading and Writing Compressed Datafiles

### Problem

You need to read or write data in a file with `gzip` or `bz2` compression.

## Solution

The `gzip` and `bz2` modules make it easy to work with such files. Both modules provide an alternative implementation of `open()` that can be used for this purpose. For example, to read compressed files as text, do this:

```
# gzip compression
import gzip
with gzip.open('somefile.gz', 'rt') as f:
    text = f.read()

# bz2 compression
import bz2
with bz2.open('somefile.bz2', 'rt') as f:
    text = f.read()
```

Similarly, to write compressed data, do this:

```
# gzip compression
import gzip
with gzip.open('somefile.gz', 'wt') as f:
    f.write(text)

# bz2 compression
import bz2
with bz2.open('somefile.bz2', 'wt') as f:
    f.write(text)
```

As shown, all I/O will use text and perform Unicode encoding/decoding. If you want to work with binary data instead, use a file mode of `rb` or `wb`.

## Discussion

For the most part, reading or writing compressed data is straightforward. However, be aware that choosing the correct file mode is critically important. If you don't specify a mode, the default mode is binary, which will break programs that expect to receive text. Both `gzip.open()` and `bz2.open()` accept the same parameters as the built-in `open()` function, including encoding, errors, newline, and so forth.

When writing compressed data, the compression level can be optionally specified using the `compresslevel` keyword argument. For example:

```
with gzip.open('somefile.gz', 'wt', compresslevel=5) as f:
    f.write(text)
```

The default level is 9, which provides the highest level of compression. Lower levels offer better performance, but not as much compression.

Finally, a little-known feature of `gzip.open()` and `bz2.open()` is that they can be layered on top of an existing file opened in binary mode. For example, this works:



```
import gzip

f = open('somefile.gz', 'rb')
with gzip.open(f, 'rt') as g:
    text = g.read()
```

This allows the `gzip` and `bz2` modules to work with various file-like objects such as sockets, pipes, and in-memory files.

## 5.8. Iterating Over Fixed-Sized Records

### Problem

Instead of iterating over a file by lines, you want to iterate over a collection of fixed-sized records or chunks.

### Solution

Use the `iter()` function and `functools.partial()` using this neat trick:

```
from functools import partial

RECORD_SIZE = 32

with open('somefile.data', 'rb') as f:
    records = iter(partial(f.read, RECORD_SIZE), b'')
    for r in records:
        ...
```

The `records` object in this example is an iterable that will produce fixed-sized chunks until the end of the file is reached. However, be aware that the last item may have fewer bytes than expected if the file size is not an exact multiple of the record size.

### Discussion

A little-known feature of the `iter()` function is that it can create an iterator if you pass it a callable and a sentinel value. The resulting iterator simply calls the supplied callable over and over again until it returns the sentinel, at which point iteration stops.

In the solution, the `functools.partial` is used to create a callable that reads a fixed number of bytes from a file each time it's called. The sentinel of `b''` is what gets returned when a file is read but the end of file has been reached.

Last, but not least, the solution shows the file being opened in binary mode. For reading fixed-sized records, this would probably be the most common case. For text files, reading line by line (the default iteration behavior) is more common.

## 5.9. Reading Binary Data into a Mutable Buffer

### Problem

You want to read binary data directly into a mutable buffer without any intermediate copying. Perhaps you want to mutate the data in-place and write it back out to a file.

### Solution

To read data into a mutable array, use the `readinto()` method of files. For example:

```
import os.path

def read_into_buffer(filename):
    buf = bytearray(os.path.getsize(filename))
    with open(filename, 'rb') as f:
        f.readinto(buf)
    return buf
```

Here is an example that illustrates the usage:

```
>>> # Write a sample file
>>> with open('sample.bin', 'wb') as f:
...     f.write(b'Hello World')
...
>>> buf = read_into_buffer('sample.bin')
>>> buf
bytearray(b'Hello World')
>>> buf[0:5] = b'Hallo'
>>> buf
bytearray(b'Hallo World')
>>> with open('newsample.bin', 'wb') as f:
...     f.write(buf)
...
11
>>>
```

### Discussion

The `readinto()` method of files can be used to fill any preallocated array with data. This even includes arrays created from the `array` module or libraries such as `numpy`. Unlike the normal `read()` method, `readinto()` fills the contents of an existing buffer rather than allocating new objects and returning them. Thus, you might be able to use it to avoid making extra memory allocations. For example, if you are reading a binary file consisting of equally sized records, you can write code like this:

```
record_size = 32          # Size of each record (adjust value)

buf = bytearray(record_size)
with open('somefile', 'rb') as f:
```

```

while True:
    n = f.readinto(buf)
    if n < record_size:
        break
    # Use the contents of buf
    ...

```

Another interesting feature to use here might be a `memoryview`, which lets you make zero-copy slices of an existing buffer and even change its contents. For example:

```

>>> buf
bytearray(b'Hello World')
>>> m1 = memoryview(buf)
>>> m2 = m1[-5:]
>>> m2
<memory at 0x100681390>
>>> m2[:] = b'WORLD'
>>> buf
bytearray(b'Hello WORLD')
>>>

```

One caution with using `f.readinto()` is that you must always make sure to check its return code, which is the number of bytes actually read.

If the number of bytes is smaller than the size of the supplied buffer, it might indicate truncated or corrupted data (e.g., if you were expecting an exact number of bytes to be read).

Finally, be on the lookout for other “into” related functions in various library modules (e.g., `recv_into()`, `pack_into()`, etc.). Many other parts of Python have support for direct I/O or data access that can be used to fill or alter the contents of arrays and buffers.

See [Recipe 6.12](#) for a significantly more advanced example of interpreting binary structures and usage of `memoryviews`.

## 5.10. Memory Mapping Binary Files

### Problem

You want to memory map a binary file into a mutable byte array, possibly for random access to its contents or to make in-place modifications.

### Solution

Use the `mmap` module to memory map files. Here is a utility function that shows how to open a file and memory map it in a portable manner:

```

import os
import mmap

```

```
def memory_map(filename, access=mmap.ACCESS_WRITE):
    size = os.path.getsize(filename)
    fd = os.open(filename, os.O_RDWR)
    return mmap.mmap(fd, size, access=access)
```

To use this function, you would need to have a file already created and filled with data. Here is an example of how you could initially create a file and expand it to a desired size:

```
>>> size = 1000000
>>> with open('data', 'wb') as f:
...     f.seek(size-1)
...     f.write(b'\x00')
...
>>>
```

Now here is an example of memory mapping the contents using the `memory_map()` function:

```
>>> m = memory_map('data')
>>> len(m)
1000000
>>> m[0:10]
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
>>> m[0]
0
>>> # Reassign a slice
>>> m[0:11] = b'Hello World'
>>> m.close()

>>> # Verify that changes were made
>>> with open('data', 'rb') as f:
...     print(f.read(11))
...
b'Hello World'
>>>
```

The `mmap` object returned by `mmap()` can also be used as a context manager, in which case the underlying file is closed automatically. For example:

```
>>> with memory_map('data') as m:
...     print(len(m))
...     print(m[0:10])
...
1000000
b'Hello World'
>>> m.closed
True
>>>
```

By default, the `memory_map()` function shown opens a file for both reading and writing. Any modifications made to the data are copied back to the original file. If read-only

access is needed instead, supply `mmap.ACCESS_READ` for the `access` argument. For example:

```
m = memory_map(filename, mmap.ACCESS_READ)
```

If you intend to modify the data locally, but don't want those changes written back to the original file, use `mmap.ACCESS_COPY`:

```
m = memory_map(filename, mmap.ACCESS_COPY)
```

## Discussion

Using `mmap` to map files into memory can be an efficient and elegant means for randomly accessing the contents of a file. For example, instead of opening a file and performing various combinations of `seek()`, `read()`, and `write()` calls, you can simply map the file and access the data using slicing operations.

Normally, the memory exposed by `mmap()` looks like a `bytearray` object. However, you can interpret the data differently using a `memoryview`. For example:

```
>>> m = memory_map('data')
>>> # Memoryview of unsigned integers
>>> v = memoryview(m).cast('I')
>>> v[0] = 7
>>> m[0:4]
b'\x07\x00\x00\x00'
>>> m[0:4] = b'\x07\x01\x00\x00'
>>> v[0]
263
>>>
```

It should be emphasized that memory mapping a file does not cause the entire file to be read into memory. That is, it's not copied into some kind of memory buffer or array. Instead, the operating system merely reserves a section of virtual memory for the file contents. As you access different regions, those portions of the file will be read and mapped into the memory region as needed. However, parts of the file that are never accessed simply stay on disk. This all happens transparently, behind the scenes.

If more than one Python interpreter memory maps the same file, the resulting `mmap` object can be used to exchange data between interpreters. That is, all interpreters can read/write data simultaneously, and changes made to the data in one interpreter will automatically appear in the others. Obviously, some extra care is required to synchronize things, but this kind of approach is sometimes used as an alternative to transmitting data in messages over pipes or sockets.

As shown, this recipe has been written to be as general purpose as possible, working on both Unix and Windows. Be aware that there are some platform differences concerning the use of the `mmap()` call hidden behind the scenes. In addition, there are options to

create anonymously mapped memory regions. If this is of interest to you, make sure you carefully read the Python documentation [on the subject](#).

## 5.11. Manipulating Pathnames

### Problem

You need to manipulate pathnames in order to find the base filename, directory name, absolute path, and so on.

### Solution

To manipulate pathnames, use the functions in the `os.path` module. Here is an interactive example that illustrates a few key features:

```
>>> import os
>>> path = '/Users/beazley/Data/data.csv'

>>> # Get the last component of the path
>>> os.path.basename(path)
'data.csv'

>>> # Get the directory name
>>> os.path.dirname(path)
'/Users/beazley/Data'

>>> # Join path components together
>>> os.path.join('tmp', 'data', os.path.basename(path))
'tmp/data/data.csv'

>>> # Expand the user's home directory
>>> path = '~/Data/data.csv'
>>> os.path.expanduser(path)
'/Users/beazley/Data/data.csv'

>>> # Split the file extension
>>> os.path.splitext(path)
('~/Data/data', '.csv')
>>>
```

### Discussion

For any manipulation of filenames, you should use the `os.path` module instead of trying to cook up your own code using the standard string operations. In part, this is for portability. The `os.path` module knows about differences between Unix and Windows and can reliably deal with filenames such as *Data/data.csv* and *Data\data.csv*. Second, you really shouldn't spend your time reinventing the wheel. It's usually best to use the functionality that's already provided for you.

It should be noted that the `os.path` module has many more features not shown in this recipe. Consult the documentation for more functions related to file testing, symbolic links, and so forth.

## 5.12. Testing for the Existence of a File

### Problem

You need to test whether or not a file or directory exists.

### Solution

Use the `os.path` module to test for the existence of a file or directory. For example:

```
>>> import os
>>> os.path.exists('/etc/passwd')
True
>>> os.path.exists('/tmp/span')
False
>>>
```

You can perform further tests to see what kind of file it might be. These tests return `False` if the file in question doesn't exist:

```
>>> # Is a regular file
>>> os.path.isfile('/etc/passwd')
True

>>> # Is a directory
>>> os.path.isdir('/etc/passwd')
False

>>> # Is a symbolic link
>>> os.path.islink('/usr/local/bin/python3')
True

>>> # Get the file linked to
>>> os.path.realpath('/usr/local/bin/python3')
'/usr/local/bin/python3.3'
>>>
```

If you need to get metadata (e.g., the file size or modification date), that is also available in the `os.path` module.

```
>>> os.path.getsize('/etc/passwd')
3669
>>> os.path.getmtime('/etc/passwd')
1272478234.0
>>> import time
>>> time.ctime(os.path.getmtime('/etc/passwd'))
```

```
'Wed Apr 28 13:10:34 2010'  
>>>
```

## Discussion

File testing is a straightforward operation using `os.path`. Probably the only thing to be aware of when writing scripts is that you might need to worry about permissions—especially for operations that get metadata. For example:

```
>>> os.path.getsize('/Users/guido/Desktop/foo.txt')  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
  File "/usr/local/lib/python3.3/genericpath.py", line 49, in getsize  
    return os.stat(filename).st_size  
PermissionError: [Errno 13] Permission denied: '/Users/guido/Desktop/foo.txt'  
>>>
```

## 5.13. Getting a Directory Listing

### Problem

You want to get a list of the files contained in a directory on the filesystem.

### Solution

Use the `os.listdir()` function to obtain a list of files in a directory:

```
import os  
names = os.listdir('somedir')
```

This will give you the raw directory listing, including all files, subdirectories, symbolic links, and so forth. If you need to filter the data in some way, consider using a list comprehension combined with various functions in the `os.path` library. For example:

```
import os.path  
# Get all regular files  
names = [name for name in os.listdir('somedir')  
         if os.path.isfile(os.path.join('somedir', name))]  
  
# Get all dirs  
dirnames = [name for name in os.listdir('somedir')  
            if os.path.isdir(os.path.join('somedir', name))]
```

The `startswith()` and `endswith()` methods of strings can be useful for filtering the contents of a directory as well. For example:

```
pyfiles = [name for name in os.listdir('somedir')  
           if name.endswith('.py')]
```



For filename matching, you may want to use the `glob` or `fnmatch` modules instead. For example:

```
import glob
pyfiles = glob.glob('somedir/*.py')

from fnmatch import fnmatch
pyfiles = [name for name in os.listdir('somedir')
           if fnmatch(name, '*.py')]
```

## Discussion

Getting a directory listing is easy, but it only gives you the names of entries in the directory. If you want to get additional metadata, such as file sizes, modification dates, and so forth, you either need to use additional functions in the `os.path` module or use the `os.stat()` function. To collect the data. For example:

```
# Example of getting a directory listing

import os
import os.path
import glob

pyfiles = glob.glob('*.py')

# Get file sizes and modification dates
name_sz_date = [(name, os.path.getsize(name), os.path.getmtime(name))
                 for name in pyfiles]

for name, size, mtime in name_sz_date:
    print(name, size, mtime)

# Alternative: Get file metadata
file_metadata = [(name, os.stat(name)) for name in pyfiles]
for name, meta in file_metadata:
    print(name, meta.st_size, meta.st_mtime)
```

Last, but not least, be aware that there are subtle issues that can arise in filename handling related to encodings. Normally, the entries returned by a function such as `os.listdir()` are decoded according to the system default filename encoding. However, it's possible under certain circumstances to encounter un-decodable filenames. Recipes 5.14 and 5.15 have more details about handling such names.

## 5.14. Bypassing Filename Encoding

### Problem

You want to perform file I/O operations using raw filenames that have not been decoded or encoded according to the default filename encoding.

### Solution

By default, all filenames are encoded and decoded according to the text encoding returned by `sys.getfilesystemencoding()`. For example:

```
>>> sys.getfilesystemencoding()
'utf-8'
>>>
```

If you want to bypass this encoding for some reason, specify a filename using a raw byte string instead. For example:

```
>>> # Write a file using a unicode filename
>>> with open('jalapeño.txt', 'w') as f:
...     f.write('Spicy!')
...
6
>>> # Directory listing (decoded)
>>> import os
>>> os.listdir('.')
['jalapeño.txt']

>>> # Directory listing (raw)
>>> os.listdir(b'.') # Note: byte string
[b'jalapen\xcc\x83o.txt']

>>> # Open file with raw filename
>>> with open(b'jalapen\xcc\x83o.txt') as f:
...     print(f.read())
...
Spicy!
>>>
```

As you can see in the last two operations, the filename handling changes ever so slightly when byte strings are supplied to file-related functions, such as `open()` and `os.listdir()`.

### Discussion

Under normal circumstances, you shouldn't need to worry about filename encoding and decoding—normal filename operations should just work. However, many operating systems may allow a user through accident or malice to create files with names that don't

conform to the expected encoding rules. Such filenames may mysteriously break Python programs that work with a lot of files.

Reading directories and working with filenames as raw undecoded bytes has the potential to avoid such problems, albeit at the cost of programming convenience.

See [Recipe 5.15](#) for a recipe on printing undecodable filenames.

## 5.15. Printing Bad Filenames

### Problem

Your program received a directory listing, but when it tried to print the filenames, it crashed with a `UnicodeEncodeError` exception and a cryptic message about “surrogates not allowed.”

### Solution

When printing filenames of unknown origin, use this convention to avoid errors:

```
def bad_filename(filename):
    return repr(filename)[1:-1]

try:
    print(filename)
except UnicodeEncodeError:
    print(bad_filename(filename))
```

### Discussion

This recipe is about a potentially rare but very annoying problem regarding programs that must manipulate the filesystem. By default, Python assumes that all filenames are encoded according to the setting reported by `sys.getfilesystemencoding()`. However, certain filesystems don’t necessarily enforce this encoding restriction, thereby allowing files to be created without proper filename encoding. It’s not common, but there is always the danger that some user will do something silly and create such a file by accident (e.g., maybe passing a bad filename to `open()` in some buggy code).

When executing a command such as `os.listdir()`, bad filenames leave Python in a bind. On the one hand, it can’t just discard bad names. On the other hand, it still can’t turn the filename into a proper text string. Python’s solution to this problem is to take an undecodable byte value `\xhh` in a filename and map it into a so-called “surrogate encoding” represented by the Unicode character `\udchh`. Here is an example of how a bad directory listing might look if it contained a filename *bäd.txt*, encoded as Latin-1 instead of UTF-8:

```
>>> import os
>>> files = os.listdir('.')
>>> files
['spam.py', 'b\uudce4d.txt', 'foo.txt']
>>>
```

If you have code that manipulates filenames or even passes them to functions such as `open()`, everything works normally. It's only in situations where you want to output the filename that you run into trouble (e.g., printing it to the screen, logging it, etc.). Specifically, if you tried to print the preceding listing, your program will crash:

```
>>> for name in files:
...     print(name)
...
spam.py
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
UnicodeEncodeError: 'utf-8' codec can't encode character '\uudce4' in
position 1: surrogates not allowed
>>>
```

The reason it crashes is that the character `\uudce4` is technically invalid Unicode. It's actually the second half of a two-character combination known as a surrogate pair. However, since the first half is missing, it's invalid Unicode. Thus, the only way to produce successful output is to take corrective action when a bad filename is encountered. For example, changing the code to the recipe produces the following:

```
>>> for name in files:
...     try:
...         print(name)
...     except UnicodeEncodeError:
...         print(bad_filename(name))
...
spam.py
b\uudce4d.txt
foo.txt
>>>
```

The choice of what to do for the `bad_filename()` function is largely up to you. Another option is to re-encode the value in some way, like this:

```
def bad_filename(filename):
    temp = filename.encode(sys.getfilesystemencoding(), errors='surrogateescape')
    return temp.decode('latin-1')
```

Using this version produces the following output:

```
>>> for name in files:
...     try:
...         print(name)
...     except UnicodeEncodeError:
...         print(bad_filename(name))
...
spam.py
b\uudce4d.txt
foo.txt
>>>
```

```
spam.py
bäd.txt
foo.txt
>>>
```

This recipe will likely be ignored by most readers. However, if you're writing mission-critical scripts that need to work reliably with filenames and the filesystem, it's something to think about. Otherwise, you might find yourself called back into the office over the weekend to debug a seemingly inscrutable error.

## 5.16. Adding or Changing the Encoding of an Already Open File

### Problem

You want to add or change the Unicode encoding of an already open file without closing it first.

### Solution

If you want to add Unicode encoding/decoding to an already existing file object that's opened in binary mode, wrap it with an `io.TextIOWrapper()` object. For example:

```
import urllib.request
import io

u = urllib.request.urlopen('http://www.python.org')
f = io.TextIOWrapper(u, encoding='utf-8')
text = f.read()
```

If you want to change the encoding of an already open text-mode file, use its `detach()` method to remove the existing text encoding layer before replacing it with a new one. Here is an example of changing the encoding on `sys.stdout`:

```
>>> import sys
>>> sys.stdout.encoding
'UTF-8'
>>> sys.stdout = io.TextIOWrapper(sys.stdout.detach(), encoding='latin-1')
>>> sys.stdout.encoding
'latin-1'
>>>
```

Doing this might break the output of your terminal. It's only meant to illustrate.

### Discussion

The I/O system is built as a series of layers. You can see the layers yourself by trying this simple example involving a text file:

```

>>> f = open('sample.txt', 'w')
>>> f
<_io.TextIOWrapper name='sample.txt' mode='w' encoding='UTF-8'>
>>> f.buffer
<_io.BufferedWriter name='sample.txt'>
>>> f.buffer.raw
<_io.FileIO name='sample.txt' mode='wb'>
>>>

```

In this example, `io.TextIOWrapper` is a text-handling layer that encodes and decodes Unicode, `io.BufferedWriter` is a buffered I/O layer that handles binary data, and `io.FileIO` is a raw file representing the low-level file descriptor in the operating system. Adding or changing the text encoding involves adding or changing the topmost `io.TextIOWrapper` layer.

As a general rule, it's not safe to directly manipulate the different layers by accessing the attributes shown. For example, see what happens if you try to change the encoding using this technique:

```

>>> f
<_io.TextIOWrapper name='sample.txt' mode='w' encoding='UTF-8'>
>>> f = io.TextIOWrapper(f.buffer, encoding='latin-1')
>>> f
<_io.TextIOWrapper name='sample.txt' encoding='latin-1'>
>>> f.write('Hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
>>>

```

It doesn't work because the original value of `f` got destroyed and closed the underlying file in the process.

The `detach()` method disconnects the topmost layer of a file and returns the next lower layer. Afterward, the top layer will no longer be usable. For example:

```

>>> f = open('sample.txt', 'w')
>>> f
<_io.TextIOWrapper name='sample.txt' mode='w' encoding='UTF-8'>
>>> b = f.detach()
>>> b
<_io.BufferedWriter name='sample.txt'>
>>> f.write('hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: underlying buffer has been detached
>>>

```

Once detached, however, you can add a new top layer to the returned result. For example:

```

>>> f = io.TextIOWrapper(b, encoding='latin-1')
>>> f

```

```
<_io.TextIOWrapper name='sample.txt' encoding='latin-1'>
>>>
```

Although changing the encoding has been shown, it is also possible to use this technique to change the line handling, error policy, and other aspects of file handling. For example:

```
>>> sys.stdout = io.TextIOWrapper(sys.stdout.detach(), encoding='ascii',
...                               errors='xmlcharrefreplace')
>>> print('Jalape\u00f1o')
Jalape&#241;o
>>>
```

Notice how the non-ASCII character ñ has been replaced by &#241; in the output.

## 5.17. Writing Bytes to a Text File

### Problem

You want to write raw bytes to a file opened in text mode.

### Solution

Simply write the byte data to the files underlying buffer. For example:

```
>>> import sys
>>> sys.stdout.write(b'Hello\n')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not bytes
>>> sys.stdout.buffer.write(b'Hello\n')
Hello
5
>>>
```

Similarly, binary data can be read from a text file by reading from its buffer attribute instead.

### Discussion

The I/O system is built from layers. Text files are constructed by adding a Unicode encoding/decoding layer on top of a buffered binary-mode file. The buffer attribute simply points at this underlying file. If you access it, you'll bypass the text encoding/decoding layer.

The example involving `sys.stdout` might be viewed as a special case. By default, `sys.stdout` is always opened in text mode. However, if you are writing a script that actually needs to dump binary data to standard output, you can use the technique shown to bypass the text encoding.)

## 5.18. Wrapping an Existing File Descriptor As a File Object

### Problem

You have an integer file descriptor corresponding to an already open I/O channel on the operating system (e.g., file, pipe, socket, etc.), and you want to wrap a higher-level Python file object around it.

### Solution

A file descriptor is different than a normal open file in that it is simply an integer handle assigned by the operating system to refer to some kind of system I/O channel. If you happen to have such a file descriptor, you can wrap a Python file object around it using the `open()` function. However, you simply supply the integer file descriptor as the first argument instead of the filename. For example:

```
# Open a low-level file descriptor
import os
fd = os.open('somefile.txt', os.O_WRONLY | os.O_CREAT)

# Turn into a proper file
f = open(fd, 'wt')
f.write('hello world\n')
f.close()
```

When the high-level file object is closed or destroyed, the underlying file descriptor will also be closed. If this is not desired, supply the optional `closefd=False` argument to `open()`. For example:

```
# Create a file object, but don't close underlying fd when done
f = open(fd, 'wt', closefd=False)
...
```

### Discussion

On Unix systems, this technique of wrapping a file descriptor can be a convenient means for putting a file-like interface on an existing I/O channel that was opened in a different way (e.g., pipes, sockets, etc.). For instance, here is an example involving sockets:

```
from socket import socket, AF_INET, SOCK_STREAM

def echo_client(client_sock, addr):
    print('Got connection from', addr)

    # Make text-mode file wrappers for socket reading/writing
    client_in = open(client_sock.fileno(), 'rt', encoding='latin-1',
                     closefd=False)
    client_out = open(client_sock.fileno(), 'wt', encoding='latin-1',
                      closefd=False)
```



```

    # Echo lines back to the client using file I/O
    for line in client_in:
        client_out.write(line)
        client_out.flush()
    client_sock.close()

def echo_server(address):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(address)
    sock.listen(1)
    while True:
        client, addr = sock.accept()
        echo_client(client, addr)

```

It's important to emphasize that the above example is only meant to illustrate a feature of the built-in `open()` function and that it only works on Unix-based systems. If you are trying to put a file-like interface on a socket and need your code to be cross platform, use the `makefile()` method of sockets instead. However, if portability is not a concern, you'll find that the above solution provides much better performance than using `makefile()`.

You can also use this to make a kind of alias that allows an already open file to be used in a slightly different way than how it was first opened. For example, here's how you could create a file object that allows you to emit binary data on `stdout` (which is normally opened in text mode):

```

import sys
# Create a binary-mode file for stdout
bstdout = open(sys.stdout.fileno(), 'wb', closefd=False)
bstdout.write(b'Hello World\n')
bstdout.flush()

```

Although it's possible to wrap an existing file descriptor as a proper file, be aware that not all file modes may be supported and that certain kinds of file descriptors may have funny side effects (especially with respect to error handling, end-of-file conditions, etc.). The behavior can also vary according to operating system. In particular, none of the examples are likely to work on non-Unix systems. The bottom line is that you'll need to thoroughly test your implementation to make sure it works as expected.

## 5.19. Making Temporary Files and Directories

### Problem

You need to create a temporary file or directory for use when your program executes. Afterward, you possibly want the file or directory to be destroyed.

## Solution

The `tempfile` module has a variety of functions for performing this task. To make an unnamed temporary file, use `tempfile.TemporaryFile()`:

```
from tempfile import TemporaryFile

with TemporaryFile('w+t') as f:
    # Read/write to the file
    f.write('Hello World\n')
    f.write('Testing\n')

    # Seek back to beginning and read the data
    f.seek(0)
    data = f.read()

# Temporary file is destroyed
```

Or, if you prefer, you can also use the file like this:

```
f = TemporaryFile('w+t')
# Use the temporary file
...
f.close()
# File is destroyed
```

The first argument to `TemporaryFile()` is the file mode, which is usually `w+t` for text and `w+b` for binary. This mode simultaneously supports reading and writing, which is useful here since closing the file to change modes would actually destroy it. `TemporaryFile()` additionally accepts the same arguments as the built-in `open()` function. For example:

```
with TemporaryFile('w+t', encoding='utf-8', errors='ignore') as f:
    ...
```

On most Unix systems, the file created by `TemporaryFile()` is unnamed and won't even have a directory entry. If you want to relax this constraint, use `NamedTemporaryFile()` instead. For example:

```
from tempfile import NamedTemporaryFile

with NamedTemporaryFile('w+t') as f:
    print('filename is:', f.name)
    ...

# File automatically destroyed
```

Here, the `f.name` attribute of the opened file contains the filename of the temporary file. This can be useful if it needs to be given to some other code that needs to open the file. As with `TemporaryFile()`, the resulting file is automatically deleted when it's closed. If you don't want this, supply a `delete=False` keyword argument. For example:

```

with NamedTemporaryFile('w+t', delete=False) as f:
    print('filename is:', f.name)
...

```

To make a temporary directory, use `tempfile.TemporaryDirectory()`. For example:

```

from tempfile import TemporaryDirectory
with TemporaryDirectory() as dirname:
    print('dirname is:', dirname)
    # Use the directory
...
# Directory and all contents destroyed

```

## Discussion

The `TemporaryFile()`, `NamedTemporaryFile()`, and `TemporaryDirectory()` functions are probably the most convenient way to work with temporary files and directories, because they automatically handle all of the steps of creation and subsequent cleanup. At a lower level, you can also use the `mkstemp()` and `mkdtemp()` to create temporary files and directories. For example:

```

>>> import tempfile
>>> tempfile.mkstemp()
(3, '/var/folders/7W/7WZL5sfZEF0pljrEB1UMWE+++TI/-Tmp-/tmp7fefhv')
>>> tempfile.mkdtemp()
'/var/folders/7W/7WZL5sfZEF0pljrEB1UMWE+++TI/-Tmp-/tmp5wvcv6'
>>>

```

However, these functions don't really take care of further management. For example, the `mkstemp()` function simply returns a raw OS file descriptor and leaves it up to you to turn it into a proper file. Similarly, it's up to you to clean up the files if you want.

Normally, temporary files are created in the system's default location, such as `/var/tmp` or similar. To find out the actual location, use the `tempfile.gettempdir()` function. For example:

```

>>> tempfile.gettempdir()
'/var/folders/7W/7WZL5sfZEF0pljrEB1UMWE+++TI/-Tmp-'
>>>

```

All of the temporary-file-related functions allow you to override this directory as well as the naming conventions using the `prefix`, `suffix`, and `dir` keyword arguments. For example:

```

>>> f = NamedTemporaryFile(prefix='mytemp', suffix='.txt', dir='/tmp')
>>> f.name
'/tmp/mytemp8ee899.txt'
>>>

```

Last, but not least, to the extent possible, the `tempfile()` module creates temporary files in the most secure manner possible. This includes only giving access permission

to the current user and taking steps to avoid race conditions in file creation. Be aware that there can be differences between platforms. Thus, you should make sure to check [the official documentation](#) for the finer points.

## 5.20. Communicating with Serial Ports

### Problem

You want to read and write data over a serial port, typically to interact with some kind of hardware device (e.g., a robot or sensor).

### Solution

Although you can probably do this directly using Python’s built-in I/O primitives, your best bet for serial communication is to use the [pySerial package](#). Getting started with the package is very easy. You simply open up a serial port using code like this:

```
import serial
ser = serial.Serial('/dev/tty.usbmodem641', # Device name varies
                   baudrate=9600,
                   bytesize=8,
                   parity='N',
                   stopbits=1)
```

The device name will vary according to the kind of device and operating system. For instance, on Windows, you can use a device of 0, 1, and so on, to open up the communication ports such as “COM0” and “COM1.” Once open, you can read and write data using `read()`, `readline()`, and `write()` calls. For example:

```
ser.write(b'G1 X50 Y50\r\n')
resp = ser.readline()
```

For the most part, simple serial communication should be pretty simple from this point forward.

### Discussion

Although simple on the surface, serial communication can sometimes get rather messy. One reason you should use a package such as `pySerial` is that it provides support for advanced features (e.g., timeouts, control flow, buffer flushing, handshaking, etc.). For instance, if you want to enable RTS-CTS handshaking, you simply provide a `rtscts=True` argument to `Serial()`. The provided documentation is excellent, so there’s little benefit to paraphrasing it here.

Keep in mind that all I/O involving serial ports is binary. Thus, make sure you write your code to use bytes instead of text (or perform proper text encoding/decoding as

needed). The `struct` module may also be useful should you need to create binary-coded commands or packets.

## 5.21. Serializing Python Objects

### Problem

You need to serialize a Python object into a byte stream so that you can do things such as save it to a file, store it in a database, or transmit it over a network connection.

### Solution

The most common approach for serializing data is to use the `pickle` module. To dump an object to a file, you do this:

```
import pickle

data = ...    # Some Python object
f = open('somefile', 'wb')
pickle.dump(data, f)
```

To dump an object to a string, use `pickle.dumps()`:

```
s = pickle.dumps(data)
```

To re-create an object from a byte stream, use either the `pickle.load()` or `pickle.loads()` functions. For example:

```
# Restore from a file
f = open('somefile', 'rb')
data = pickle.load(f)

# Restore from a string
data = pickle.loads(s)
```

### Discussion

For most programs, usage of the `dump()` and `load()` function is all you need to effectively use `pickle`. It simply works with most Python data types and instances of user-defined classes. If you're working with any kind of library that lets you do things such as save/restore Python objects in databases or transmit objects over the network, there's a pretty good chance that `pickle` is being used.

`pickle` is a Python-specific self-describing data encoding. By self-describing, the serialized data contains information related to the start and end of each object as well as information about its type. Thus, you don't need to worry about defining records—it simply works. For example, if working with multiple objects, you can do this:

```

>>> import pickle
>>> f = open('somedata', 'wb')
>>> pickle.dump([1, 2, 3, 4], f)
>>> pickle.dump('hello', f)
>>> pickle.dump({'Apple', 'Pear', 'Banana'}, f)
>>> f.close()
>>> f = open('somedata', 'rb')
>>> pickle.load(f)
[1, 2, 3, 4]
>>> pickle.load(f)
'hello'
>>> pickle.load(f)
{'Apple', 'Pear', 'Banana'}
>>>

```

You can pickle functions, classes, and instances, but the resulting data only encodes name references to the associated code objects. For example:

```

>>> import math
>>> import pickle.
>>> pickle.dumps(math.cos)
b'\x80\x03cmath\ncos\nq\x00.'
>>>

```

When the data is unpickled, it is assumed that all of the required source is available. Modules, classes, and functions will automatically be imported as needed. For applications where Python data is being shared between interpreters on different machines, this is a potential maintenance issue, as all machines must have access to the same source code.



`pickle.load()` should never be used on untrusted data. As a side effect of loading, pickle will automatically load modules and make instances. However, an evildoer who knows how pickle works can create “malformed” data that causes Python to execute arbitrary system commands. Thus, it’s essential that pickle only be used internally with interpreters that have some ability to authenticate one another.

Certain kinds of objects can’t be pickled. These are typically objects that involve some sort of external system state, such as open files, open network connections, threads, processes, stack frames, and so forth. User-defined classes can sometimes work around these limitations by providing `__getstate__()` and `__setstate__()` methods. If defined, `pickle.dump()` will call `__getstate__()` to get an object that can be pickled. Similarly, `__setstate__()` will be invoked on unpickling. To illustrate what’s possible, here is a class that internally defines a thread but can still be pickled/unpickled:

```

# countdown.py
import time
import threading

```

```

class Countdown:
    def __init__(self, n):
        self.n = n
        self.thr = threading.Thread(target=self.run)
        self.thr.daemon = True
        self.thr.start()

    def run(self):
        while self.n > 0:
            print('T-minus', self.n)
            self.n -= 1
            time.sleep(5)

    def __getstate__(self):
        return self.n

    def __setstate__(self, n):
        self.__init__(n)

```

Try the following experiment involving pickling:

```

>>> import countdown
>>> c = countdown.Countdown(30)
>>> T-minus 30
T-minus 29
T-minus 28
...

>>> # After a few moments
>>> f = open('cstate.p', 'wb')
>>> import pickle
>>> pickle.dump(c, f)
>>> f.close()

```

Now quit Python and try this after restart:

```

>>> f = open('cstate.p', 'rb')
>>> pickle.load(f)
countdown.Countdown object at 0x10069e2d0>
T-minus 19
T-minus 18
...

```

You should see the thread magically spring to life again, picking up where it left off when you first pickled it.

`pickle` is not a particularly efficient encoding for large data structures such as binary arrays created by libraries like the `array` module or `numpy`. If you're moving large amounts of array data around, you may be better off simply saving bulk array data in a file or using a more standardized encoding, such as `HDF5` (supported by third-party libraries).

Because of its Python-specific nature and attachment to source code, you probably shouldn't use `pickle` as a format for long-term storage. For example, if the source code changes, all of your stored data might break and become unreadable. Frankly, for storing data in databases and archival storage, you're probably better off using a more standard data encoding, such as XML, CSV, or JSON. These encodings are more standardized, supported by many different languages, and more likely to be better adapted to changes in your source code.

Last, but not least, be aware that `pickle` has a huge variety of options and tricky corner cases. For the most common uses, you don't need to worry about them, but a look at the [official documentation](#) should be required if you're going to build a significant application that uses `pickle` for serialization.



---

# Data Encoding and Processing

The main focus of this chapter is using Python to process data presented in different kinds of common encodings, such as CSV files, JSON, XML, and binary packed records. Unlike the chapter on data structures, this chapter is not focused on specific algorithms, but instead on the problem of getting data in and out of a program.

## 6.1. Reading and Writing CSV Data

### Problem

You want to read or write data encoded as a CSV file.

### Solution

For most kinds of CSV data, use the `csv` library. For example, suppose you have some stock market data in a file named *stocks.csv* like this:

```
Symbol,Price,Date,Time,Change,Volume
"AA",39.48,"6/11/2007","9:36am",-0.18,181800
"AIG",71.38,"6/11/2007","9:36am",-0.15,195500
"AXP",62.58,"6/11/2007","9:36am",-0.46,935000
"BA",98.31,"6/11/2007","9:36am",+0.12,104800
"C",53.08,"6/11/2007","9:36am",-0.25,360900
"CAT",78.29,"6/11/2007","9:36am",-0.23,225400
```

Here's how you would read the data as a sequence of tuples:

```
import csv
with open('stocks.csv') as f:
    f_csv = csv.reader(f)
    headers = next(f_csv)
    for row in f_csv:
```

```
# Process row
...
```

In the preceding code, `row` will be a tuple. Thus, to access certain fields, you will need to use indexing, such as `row[0]` (Symbol) and `row[4]` (Change).

Since such indexing can often be confusing, this is one place where you might want to consider the use of named tuples. For example:

```
from collections import namedtuple
with open('stock.csv') as f:
    f_csv = csv.reader(f)
    headings = next(f_csv)
    Row = namedtuple('Row', headings)
    for r in f_csv:
        row = Row(*r)
        # Process row
    ...
```

This would allow you to use the column headers such as `row.Symbol` and `row.Change` instead of indices. It should be noted that this only works if the column headers are valid Python identifiers. If not, you might have to massage the initial headings (e.g., replacing nonidentifier characters with underscores or similar).

Another alternative is to read the data as a sequence of dictionaries instead. To do that, use this code:

```
import csv
with open('stocks.csv') as f:
    f_csv = csv.DictReader(f)
    for row in f_csv:
        # process row
    ...
```

In this version, you would access the elements of each row using the row headers. For example, `row['Symbol']` or `row['Change']`.

To write CSV data, you also use the `csv` module but create a writer object. For example:

```
headers = ['Symbol', 'Price', 'Date', 'Time', 'Change', 'Volume']
rows = [(('AA', 39.48, '6/11/2007', '9:36am', -0.18, 181800),
        ('AIG', 71.38, '6/11/2007', '9:36am', -0.15, 195500),
        ('AXP', 62.58, '6/11/2007', '9:36am', -0.46, 935000)),
        ]

with open('stocks.csv', 'w') as f:
    f_csv = csv.writer(f)
    f_csv.writerow(headers)
    f_csv.writerows(rows)
```

If you have the data as a sequence of dictionaries, do this:

```

headers = ['Symbol', 'Price', 'Date', 'Time', 'Change', 'Volume']
rows = [{ 'Symbol': 'AA', 'Price': 39.48, 'Date': '6/11/2007',
          'Time': '9:36am', 'Change': -0.18, 'Volume': 181800 },
        { 'Symbol': 'AIG', 'Price': 71.38, 'Date': '6/11/2007',
          'Time': '9:36am', 'Change': -0.15, 'Volume': 195500 },
        { 'Symbol': 'AXP', 'Price': 62.58, 'Date': '6/11/2007',
          'Time': '9:36am', 'Change': -0.46, 'Volume': 935000 },
        ]

with open('stocks.csv', 'w') as f:
    f_csv = csv.DictWriter(f, headers)
    f_csv.writeheader()
    f_csv.writerows(rows)

```

## Discussion

You should almost always prefer the use of the `csv` module over manually trying to split and parse CSV data yourself. For instance, you might be inclined to just write some code like this:

```

with open('stocks.csv') as f:
    for line in f:
        row = line.split(',')
        # process row
    ...

```

The problem with this approach is that you'll still need to deal with some nasty details. For example, if any of the fields are surrounded by quotes, you'll have to strip the quotes. In addition, if a quoted field happens to contain a comma, the code will break by producing a row with the wrong size.

By default, the `csv` library is programmed to understand CSV encoding rules used by Microsoft Excel. This is probably the most common variant, and will likely give you the best compatibility. However, if you consult the documentation for `csv`, you'll see a few ways to tweak the encoding to different formats (e.g., changing the separator character, etc.). For example, if you want to read tab-delimited data instead, use this:

```

# Example of reading tab-separated values
with open('stock.tsv') as f:
    f_tsv = csv.reader(f, delimiter='\t')
    for row in f_tsv:
        # Process row
    ...

```

If you're reading CSV data and converting it into named tuples, you need to be a little careful with validating column headers. For example, a CSV file could have a header line containing nonvalid identifier characters like this:

```

Street Address,Num-Premises,Latitude,Longitude
5412 N CLARK,10,41.980262,-87.668452

```

This will actually cause the creation of a `namedtuple` to fail with a `ValueError` exception. To work around this, you might have to scrub the headers first. For instance, carrying a regex substitution on nonvalid identifier characters like this:

```
import re
with open('stock.csv') as f:
    f_csv = csv.reader(f)
    headers = [ re.sub('[^a-zA-Z_]', '_', h) for h in next(f_csv) ]
    Row = namedtuple('Row', headers)
    for r in f_csv:
        row = Row(*r)
        # Process row
    ...
```

It's also important to emphasize that `csv` does not try to interpret the data or convert it to a type other than a string. If such conversions are important, that is something you'll need to do yourself. Here is one example of performing extra type conversions on CSV data:

```
col_types = [str, float, str, str, float, int]
with open('stocks.csv') as f:
    f_csv = csv.reader(f)
    headers = next(f_csv)
    for row in f_csv:
        # Apply conversions to the row items
        row = tuple(convert(value) for convert, value in zip(col_types, row))
    ...
```

Alternatively, here is an example of converting selected fields of dictionaries:

```
print('Reading as dicts with type conversion')
field_types = [ ('Price', float),
                ('Change', float),
                ('Volume', int) ]

with open('stocks.csv') as f:
    for row in csv.DictReader(f):
        row.update((key, conversion(row[key]))
                   for key, conversion in field_types)
    print(row)
```

In general, you'll probably want to be a bit careful with such conversions, though. In the real world, it's common for CSV files to have missing values, corrupted data, and other issues that would break type conversions. So, unless your data is guaranteed to be error free, that's something you'll need to consider (you might need to add suitable exception handling).

Finally, if your goal in reading CSV data is to perform data analysis and statistics, you might want to look at the **Pandas package**. Pandas includes a convenient `pandas.read_csv()` function that will load CSV data into a `DataFrame` object. From there,

you can generate various summary statistics, filter the data, and perform other kinds of high-level operations. An example is given in [Recipe 6.13](#).

## 6.2. Reading and Writing JSON Data

### Problem

You want to read or write data encoded as JSON (JavaScript Object Notation).

### Solution

The `json` module provides an easy way to encode and decode data in JSON. The two main functions are `json.dumps()` and `json.loads()`, mirroring the interface used in other serialization libraries, such as `pickle`. Here is how you turn a Python data structure into JSON:

```
import json

data = {
    'name' : 'ACME',
    'shares' : 100,
    'price' : 542.23
}

json_str = json.dumps(data)
```

Here is how you turn a JSON-encoded string back into a Python data structure:

```
data = json.loads(json_str)
```

If you are working with files instead of strings, you can alternatively use `json.dump()` and `json.load()` to encode and decode JSON data. For example:

```
# Writing JSON data
with open('data.json', 'w') as f:
    json.dump(data, f)

# Reading data back
with open('data.json', 'r') as f:
    data = json.load(f)
```

### Discussion

JSON encoding supports the basic types of `None`, `bool`, `int`, `float`, and `str`, as well as lists, tuples, and dictionaries containing those types. For dictionaries, keys are assumed to be strings (any nonstring keys in a dictionary are converted to strings when encoding). To be compliant with the JSON specification, you should only encode Python lists and

dictionaries. Moreover, in web applications, it is standard practice for the top-level object to be a dictionary.

The format of JSON encoding is almost identical to Python syntax except for a few minor changes. For instance, `True` is mapped to `true`, `False` is mapped to `false`, and `None` is mapped to `null`. Here is an example that shows what the encoding looks like:

```
>>> json.dumps(False)
'false'
>>> d = {'a': True,
...      'b': 'Hello',
...      'c': None}
>>> json.dumps(d)
'{"b": "Hello", "c": null, "a": true}'
>>>
```

If you are trying to examine data you have decoded from JSON, it can often be hard to ascertain its structure simply by printing it out—especially if the data contains a deep level of nested structures or a lot of fields. To assist with this, consider using the `pprint()` function in the `pprint` module. This will alphabetize the keys and output a dictionary in a more sane way. Here is an example that illustrates how you would pretty print the results of a search on Twitter:

```
>>> from urllib.request import urlopen
>>> import json
>>> u = urlopen('http://search.twitter.com/search.json?q=python&rpp=5')
>>> resp = json.loads(u.read().decode('utf-8'))
>>> from pprint import pprint
>>> pprint(resp)
{'completed_in': 0.074,
 'max_id': 264043230692245504,
 'max_id_str': '264043230692245504',
 'next_page': '?page=2&max_id=264043230692245504&q=python&rpp=5',
 'page': 1,
 'query': 'python',
 'refresh_url': '?since_id=264043230692245504&q=python',
 'results': [{'created_at': 'Thu, 01 Nov 2012 16:36:26 +0000',
               'from_user': ...
             },
             {'created_at': 'Thu, 01 Nov 2012 16:36:14 +0000',
               'from_user': ...
             },
             {'created_at': 'Thu, 01 Nov 2012 16:36:13 +0000',
               'from_user': ...
             },
             {'created_at': 'Thu, 01 Nov 2012 16:36:07 +0000',
               'from_user': ...
             },
             {'created_at': 'Thu, 01 Nov 2012 16:36:04 +0000',
               'from_user': ...
             }
            ],
}
```

```
'results_per_page': 5,
'since_id': 0,
'since_id_str': '0'}
>>>
```

Normally, JSON decoding will create dicts or lists from the supplied data. If you want to create different kinds of objects, supply the `object_pairs_hook` or `object_hook` to `json.loads()`. For example, here is how you would decode JSON data, preserving its order in an `OrderedDict`:

```
>>> s = '{"name": "ACME", "shares": 50, "price": 490.1}'
>>> from collections import OrderedDict
>>> data = json.loads(s, object_pairs_hook=OrderedDict)
>>> data
OrderedDict([('name', 'ACME'), ('shares', 50), ('price', 490.1)])
>>>
```

Here is how you could turn a JSON dictionary into a Python object:

```
>>> class JSONObject:
...     def __init__(self, d):
...         self.__dict__ = d
...
>>>
>>> data = json.loads(s, object_hook=JSONObject)
>>> data.name
'ACME'
>>> data.shares
50
>>> data.price
490.1
>>>
```

In this last example, the dictionary created by decoding the JSON data is passed as a single argument to `__init__()`. From there, you are free to use it as you will, such as using it directly as the instance dictionary of the object.

There are a few options that can be useful for encoding JSON. If you would like the output to be nicely formatted, you can use the `indent` argument to `json.dumps()`. This causes the output to be pretty printed in a format similar to that with the `pprint()` function. For example:

```
>>> print(json.dumps(data))
{"price": 542.23, "name": "ACME", "shares": 100}
>>> print(json.dumps(data, indent=4))
{
    "price": 542.23,
    "name": "ACME",
    "shares": 100
}
>>>
```

If you want the keys to be sorted on output, used the `sort_keys` argument:

```
>>> print(json.dumps(data, sort_keys=True))
{"name": "ACME", "price": 542.23, "shares": 100}
>>>
```

Instances are not normally serializable as JSON. For example:

```
>>> class Point:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...
>>> p = Point(2, 3)
>>> json.dumps(p)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.3/json/__init__.py", line 226, in dumps
    return _default_encoder.encode(obj)
  File "/usr/local/lib/python3.3/json/encoder.py", line 187, in encode
    chunks = self.iterencode(o, _one_shot=True)
  File "/usr/local/lib/python3.3/json/encoder.py", line 245, in iterencode
    return _iterencode(o, 0)
  File "/usr/local/lib/python3.3/json/encoder.py", line 169, in default
    raise TypeError(repr(o) + " is not JSON serializable")
TypeError: <__main__.Point object at 0x1006f2650> is not JSON serializable
>>>
```

If you want to serialize instances, you can supply a function that takes an instance as input and returns a dictionary that can be serialized. For example:

```
def serialize_instance(obj):
    d = { '__classname__' : type(obj).__name__ }
    d.update(vars(obj))
    return d
```

If you want to get an instance back, you could write code like this:

```
# Dictionary mapping names to known classes
classes = {
    'Point' : Point
}

def unserialize_object(d):
    clsname = d.pop('__classname__', None)
    if clsname:
        cls = classes[clsname]
        obj = cls.__new__(cls) # Make instance without calling __init__
        for key, value in d.items():
            setattr(obj, key, value)
        return obj
    else:
        return d
```



Here is an example of how these functions are used:

```
>>> p = Point(2,3)
>>> s = json.dumps(p, default=serialize_instance)
>>> s
'{"__classname__": "Point", "y": 3, "x": 2}'
>>> a = json.loads(s, object_hook=unserialize_object)
>>> a
<__main__.Point object at 0x1017577d0>
>>> a.x
2
>>> a.y
3
>>>
```

The `json` module has a variety of other options for controlling the low-level interpretation of numbers, special values such as NaN, and more. Consult the [documentation](#) for further details.

## 6.3. Parsing Simple XML Data

### Problem

You would like to extract data from a simple XML document.

### Solution

The `xml.etree.ElementTree` module can be used to extract data from simple XML documents. To illustrate, suppose you want to parse and make a summary of the RSS feed on [Planet Python](#). Here is a script that will do it:

```
from urllib.request import urlopen
from xml.etree.ElementTree import parse

# Download the RSS feed and parse it
u = urlopen('http://planet.python.org/rss20.xml')
doc = parse(u)

# Extract and output tags of interest
for item in doc.iterfind('channel/item'):
    title = item.findtext('title')
    date = item.findtext('pubDate')
    link = item.findtext('link')

    print(title)
    print(date)
    print(link)
    print()
```

If you run the preceding script, the output looks similar to the following:

```
Steve Holden: Python for Data Analysis
Mon, 19 Nov 2012 02:13:51 +0000
http://holdenweb.blogspot.com/2012/11/python-for-data-analysis.html

Vasudev Ram: The Python Data model (for v2 and v3)
Sun, 18 Nov 2012 22:06:47 +0000
http://jugad2.blogspot.com/2012/11/the-python-data-model.html

Python Diary: Been playing around with Object Databases
Sun, 18 Nov 2012 20:40:29 +0000
http://www.pythondiary.com/blog/Nov.18,2012/been-...-object-databases.html

Vasudev Ram: Wakari, Scientific Python in the cloud
Sun, 18 Nov 2012 20:19:41 +0000
http://jugad2.blogspot.com/2012/11/wakari-scientific-python-in-cloud.html

Jesse Jiryu Davis: Toro: synchronization primitives for Tornado coroutines
Sun, 18 Nov 2012 20:17:49 +0000
http://feedproxy.google.com/~r/EmptysquarePython/~3/_DOZT2Kd0hQ/
```

Obviously, if you want to do more processing, you need to replace the `print()` statements with something more interesting.

## Discussion

Working with data encoded as XML is commonplace in many applications. Not only is XML widely used as a format for exchanging data on the Internet, it is a common format for storing application data (e.g., word processing, music libraries, etc.). The discussion that follows already assumes the reader is familiar with XML basics.

In many cases, when XML is simply being used to store data, the document structure is compact and straightforward. For example, the RSS feed from the example looks similar to the following:

```
<?xml version="1.0"?>
<rss version="2.0" xmlns:dc="http://purl.org/dc/elements/1.1/">
<channel>
  <title>Planet Python</title>
  <link>http://planet.python.org/</link>
  <language>en</language>
  <description>Planet Python - http://planet.python.org/</description>
  <item>
    <title>Steve Holden: Python for Data Analysis</title>
    <guid>http://holdenweb.blogspot.com/...-data-analysis.html</guid>
    <link>http://holdenweb.blogspot.com/...-data-analysis.html</link>
    <description>...</description>
    <pubDate>Mon, 19 Nov 2012 02:13:51 +0000</pubDate>
  </item>
</channel>
```

```

<title>Vasudev Ram: The Python Data model (for v2 and v3)</title>
<guid>http://jugad2.blogspot.com/...-data-model.html</guid>
<link>http://jugad2.blogspot.com/...-data-model.html</link>
<description>...</description>
<pubDate>Sun, 18 Nov 2012 22:06:47 +0000</pubDate>
</item>
<item>
<title>Python Diary: Been playing around with Object Databases</title>
<guid>http://www.pythondiary.com/...-object-databases.html</guid>
<link>http://www.pythondiary.com/...-object-databases.html</link>
<description>...</description>
<pubDate>Sun, 18 Nov 2012 20:40:29 +0000</pubDate>
</item>
...
</channel>
</rss>

```

The `xml.etree.ElementTree.parse()` function parses the entire XML document into a document object. From there, you use methods such as `find()`, `iterfind()`, and `findtext()` to search for specific XML elements. The arguments to these functions are the names of a specific tag, such as `channel/item` or `title`.

When specifying tags, you need to take the overall document structure into account. Each find operation takes place relative to a starting element. Likewise, the tagname that you supply to each operation is also relative to the start. In the example, the call to `doc.iterfind('channel/item')` looks for all “item” elements under a “channel” element. `doc` represents the top of the document (the top-level “rss” element). The later calls to `item.findtext()` take place relative to the found “item” elements.

Each element represented by the `ElementTree` module has a few essential attributes and methods that are useful when parsing. The `tag` attribute contains the name of the tag, the `text` attribute contains enclosed text, and the `get()` method can be used to extract attributes (if any). For example:

```

>>> doc
<xml.etree.ElementTree.ElementTree object at 0x101339510>
>>> e = doc.find('channel/title')
>>> e
<Element 'title' at 0x10135b310>
>>> e.tag
'title'
>>> e.text
'Planet Python'
>>> e.get('some_attribute')
>>>

```

It should be noted that `xml.etree.ElementTree` is not the only option for XML parsing. For more advanced applications, you might consider `lxml`. It uses the same programming interface as `ElementTree`, so the example shown in this recipe works in the same

manner. You simply need to change the first import to `from lxml.etree import parse`. `lxml` provides the benefit of being fully compliant with XML standards. It is also extremely fast, and provides support for features such as validation, XSLT, and XPath.

## 6.4. Parsing Huge XML Files Incrementally

### Problem

You need to extract data from a huge XML document using as little memory as possible.

### Solution

Any time you are faced with the problem of incremental data processing, you should think of iterators and generators. Here is a simple function that can be used to incrementally process huge XML files using a very small memory footprint:

```
from lxml.etree.ElementTree import iterparse

def parse_and_remove(filename, path):
    path_parts = path.split('/')
    doc = iterparse(filename, ('start', 'end'))
    # Skip the root element
    next(doc)

    tag_stack = []
    elem_stack = []
    for event, elem in doc:
        if event == 'start':
            tag_stack.append(elem.tag)
            elem_stack.append(elem)
        elif event == 'end':
            if tag_stack == path_parts:
                yield elem
                elem_stack[-2].remove(elem)
            try:
                tag_stack.pop()
                elem_stack.pop()
            except IndexError:
                pass
```

To test the function, you now need to find a large XML file to work with. You can often find such files on government and open data websites. For example, you can download [Chicago's pothole database as XML](#). At the time of this writing, the downloaded file consists of more than 100,000 rows of data, which are encoded like this:

```
<response>
  <row>
    <row ...>
      <creation_date>2012-11-18T00:00:00</creation_date>
```

```

<status>Completed</status>
<completion_date>2012-11-18T00:00:00</completion_date>
<service_request_number>12-01906549</service_request_number>
<type_of_service_request>Pot Hole in Street</type_of_service_request>
<current_activity>Final Outcome</current_activity>
<most_recent_action>CDOT Street Cut ... Outcome</most_recent_action>
<street_address>4714 S TALMAN AVE</street_address>
<zip>60632</zip>
<x_coordinate>1159494.68618856</x_coordinate>
<y_coordinate>1873313.83503384</y_coordinate>
<ward>14</ward>
<police_district>9</police_district>
<community_area>58</community_area>
<latitude>41.808090232127896</latitude>
<longitude>-87.69053684711305</longitude>
<location latitude="41.808090232127896"
        longitude="-87.69053684711305" />
</row>
<row ...>
  <creation_date>2012-11-18T00:00:00</creation_date>
  <status>Completed</status>
  <completion_date>2012-11-18T00:00:00</completion_date>
  <service_request_number>12-01906695</service_request_number>
  <type_of_service_request>Pot Hole in Street</type_of_service_request>
  <current_activity>Final Outcome</current_activity>
  <most_recent_action>CDOT Street Cut ... Outcome</most_recent_action>
  <street_address>3510 W NORTH AVE</street_address>
  <zip>60647</zip>
  <x_coordinate>1152732.14127696</x_coordinate>
  <y_coordinate>1910409.38979075</y_coordinate>
  <ward>26</ward>
  <police_district>14</police_district>
  <community_area>23</community_area>
  <latitude>41.91002084292946</latitude>
  <longitude>-87.71435952353961</longitude>
  <location latitude="41.91002084292946"
        longitude="-87.71435952353961" />
</row>
</row>
</response>

```

Suppose you want to write a script that ranks ZIP codes by the number of pothole reports. To do it, you could write code like this:

```

from xml.etree.ElementTree import parse
from collections import Counter

potholes_by_zip = Counter()

doc = parse('potholes.xml')
for pothole in doc.iterfind('row/row'):
    potholes_by_zip[pothole.findtext('zip')] += 1

```

```

for zipcode, num in potholes_by_zip.most_common():
    print(zipcode, num)

```

The only problem with this script is that it reads and parses the entire XML file into memory. On our machine, it takes about 450 MB of memory to run. Using this recipe's code, the program changes only slightly:

```

from collections import Counter
potholes_by_zip = Counter()

data = parse_and_remove('potholes.xml', 'row/row')
for pothole in data:
    potholes_by_zip[pothole.findtext('zip')] += 1

for zipcode, num in potholes_by_zip.most_common():
    print(zipcode, num)

```

This version of code runs with a memory footprint of only 7 MB—a huge savings!

## Discussion

This recipe relies on two core features of the `ElementTree` module. First, the `iterparse()` method allows incremental processing of XML documents. To use it, you supply the filename along with an event list consisting of one or more of the following: `start`, `end`, `start-ns`, and `end-ns`. The iterator created by `iterparse()` produces tuples of the form `(event, elem)`, where `event` is one of the listed events and `elem` is the resulting XML element. For example:

```

>>> data = iterparse('potholes.xml', ('start', 'end'))
>>> next(data)
('start', <Element 'response' at 0x100771d60>)
>>> next(data)
('start', <Element 'row' at 0x100771e68>)
>>> next(data)
('start', <Element 'row' at 0x100771fc8>)
>>> next(data)
('start', <Element 'creation_date' at 0x100771f18>)
>>> next(data)
('end', <Element 'creation_date' at 0x100771f18>)
>>> next(data)
('start', <Element 'status' at 0x1006a7f18>)
>>> next(data)
('end', <Element 'status' at 0x1006a7f18>)
>>>

```

`start` events are created when an element is first created but not yet populated with any other data (e.g., child elements). `end` events are created when an element is completed. Although not shown in this recipe, `start-ns` and `end-ns` events are used to handle XML namespace declarations.

In this recipe, the start and end events are used to manage stacks of elements and tags. The stacks represent the current hierarchical structure of the document as it's being parsed, and are also used to determine if an element matches the requested path given to the `parse_and_remove()` function. If a match is made, `yield` is used to emit it back to the caller.

The following statement after the `yield` is the core feature of `ElementTree` that makes this recipe save memory:

```
elem_stack[-2].remove(elem)
```

This statement causes the previously yielded element to be removed from its parent. Assuming that no references are left to it anywhere else, the element is destroyed and memory reclaimed.

The end effect of the iterative parse and the removal of nodes is a highly efficient incremental sweep over the document. At no point is a complete document tree ever constructed. Yet, it is still possible to write code that processes the XML data in a straightforward manner.

The primary downside to this recipe is its runtime performance. When tested, the version of code that reads the entire document into memory first runs approximately twice as fast as the version that processes it incrementally. However, it requires more than 60 times as much memory. So, if memory use is a greater concern, the incremental version is a big win.

## 6.5. Turning a Dictionary into XML

### Problem

You want to take the data in a Python dictionary and turn it into XML.

### Solution

Although the `xml.etree.ElementTree` library is commonly used for parsing, it can also be used to create XML documents. For example, consider this function:

```
from xml.etree.ElementTree import Element

def dict_to_xml(tag, d):
    """
    Turn a simple dict of key/value pairs into XML
    """
    elem = Element(tag)
    for key, val in d.items():
        child = Element(key)
        child.text = str(val)
```

```

        elem.append(child)
    return elem

```

Here is an example:

```

>>> s = { 'name': 'GOOG', 'shares': 100, 'price':490.1 }
>>> e = dict_to_xml('stock', s)
>>> e
<Element 'stock' at 0x1004b64c8>
>>>

```

The result of this conversion is an `Element` instance. For I/O, it is easy to convert this to a byte string using the `tostring()` function in `xml.etree.ElementTree`. For example:

```

>>> from xml.etree.ElementTree import tostring
>>> tostring(e)
b'<stock><price>490.1</price><shares>100</shares><name>GOOG</name></stock>'
>>>

```

If you want to attach attributes to an element, use its `set()` method:

```

>>> e.set('_id', '1234')
>>> tostring(e)
b'<stock _id="1234"><price>490.1</price><shares>100</shares><name>GOOG</name></stock>'
>>>

```

If the order of the elements matters, consider making an `OrderedDict` instead of a normal dictionary. See [Recipe 1.7](#).

## Discussion

When creating XML, you might be inclined to just make strings instead. For example:

```

def dict_to_xml_str(tag, d):
    '''
    Turn a simple dict of key/value pairs into XML
    '''
    parts = ['<{}>'.format(tag)]
    for key, val in d.items():
        parts.append('<{}>{1}</{}>'.format(key, val))
    parts.append('</{}>'.format(tag))
    return ''.join(parts)

```

The problem is that you're going to make a real mess for yourself if you try to do things manually. For example, what happens if the dictionary values contain special characters like this?

```

>>> d = { 'name' : '<spam>' }

>>> # String creation
>>> dict_to_xml_str('item', d)

```



```
'<item><name><spam></name></item>'

>>> # Proper XML creation
>>> e = dict_to_xml('item',d)
>>> tostring(e)
b'<item><name>&lt;spam&gt;</name></item>'
>>>
```

Notice how in the latter example, the characters < and > got replaced with &lt; and &gt;.

Just for reference, if you ever need to manually escape or unescape such characters, you can use the `escape()` and `unescape()` functions in `xml.sax.saxutils`. For example:

```
>>> from xml.sax.saxutils import escape, unescape
>>> escape('<spam>')
'&lt;spam&gt;'
>>> unescape(_)
'<spam>'
>>>
```

Aside from creating correct output, the other reason why it's a good idea to create `Element` instances instead of strings is that they can be more easily combined together to make a larger document. The resulting `Element` instances can also be processed in various ways without ever having to worry about parsing the XML text. Essentially, you can do all of the processing of the data in a more high-level form and then output it as a string at the very end.

## 6.6. Parsing, Modifying, and Rewriting XML

### Problem

You want to read an XML document, make changes to it, and then write it back out as XML.

### Solution

The `xml.etree.ElementTree` module makes it easy to perform such tasks. Essentially, you start out by parsing the document in the usual way. For example, suppose you have a document named *pred.xml* that looks like this:

```
<?xml version="1.0"?>
<stop>
  <id>14791</id>
  <nm>Clark &amp; Balmoral</nm>
  <sri>
    <rt>22</rt>
    <d>North Bound</d>
    <dd>North Bound</dd>
  </sri>
```

```

<cr>22</cr>
<pre>
  <pt>5 MIN</pt>
  <fd>Howard</fd>
  <v>1378</v>
  <rn>22</rn>
</pre>
<pre>
  <pt>15 MIN</pt>
  <fd>Howard</fd>
  <v>1867</v>
  <rn>22</rn>
</pre>
</stop>

```

Here is an example of using ElementTree to read it and make changes to the structure:

```

>>> from xml.etree.ElementTree import parse, Element
>>> doc = parse('pred.xml')
>>> root = doc.getroot()
>>> root
<Element 'stop' at 0x100770cb0>

>>> # Remove a few elements
>>> root.remove(root.find('sri'))
>>> root.remove(root.find('cr'))

>>> # Insert a new element after <nm>...</nm>
>>> root.getchildren().index(root.find('nm'))
1
>>> e = Element('spam')
>>> e.text = 'This is a test'
>>> root.insert(2, e)

>>> # Write back to a file
>>> doc.write('newpred.xml', xml_declaration=True)
>>>

```

The result of these operations is a new XML file that looks like this:

```

<?xml version='1.0' encoding='us-ascii'?>
<stop>
  <id>14791</id>
  <nm>Clark & Balmoral</nm>
  <spam>This is a test</spam><pre>
    <pt>5 MIN</pt>
    <fd>Howard</fd>
    <v>1378</v>
    <rn>22</rn>
  </pre>
  <pre>
    <pt>15 MIN</pt>
    <fd>Howard</fd>

```

```

        <v>1867</v>
        <rn>22</rn>
    </pre>
</stop>

```

## Discussion

Modifying the structure of an XML document is straightforward, but you must remember that all modifications are generally made to the parent element, treating it as if it were a list. For example, if you remove an element, it is removed from its immediate parent using the parent's `remove()` method. If you insert or append new elements, you also use `insert()` and `append()` methods on the parent. Elements can also be manipulated using indexing and slicing operations, such as `element[i]` or `element[i:j]`.

If you need to make new elements, use the `Element` class, as shown in this recipe's solution. This is described further in [Recipe 6.5](#).

## 6.7. Parsing XML Documents with Namespaces

### Problem

You need to parse an XML document, but it's using XML namespaces.

### Solution

Consider a document that uses namespaces like this:

```

<?xml version="1.0" encoding="utf-8"?>
<top>
  <author>David Beazley</author>
  <content>
    <html xmlns="http://www.w3.org/1999/xhtml">
      <head>
        <title>Hello World</title>
      </head>
      <body>
        <h1>Hello World!</h1>
      </body>
    </html>
  </content>
</top>

```

If you parse this document and try to perform the usual queries, you'll find that it doesn't work so easily because everything becomes incredibly verbose:

```

>>> # Some queries that work
>>> doc.findtext('author')
'David Beazley'
>>> doc.find('content')

```

```

<Element 'content' at 0x100776ec0>

>>> # A query involving a namespace (doesn't work)
>>> doc.find('content/html')

>>> # Works if fully qualified
>>> doc.find('content/{http://www.w3.org/1999/xhtml}html')
<Element '{http://www.w3.org/1999/xhtml}html' at 0x1007767e0>

>>> # Doesn't work
>>> doc.findtext('content/{http://www.w3.org/1999/xhtml}html/head/title')

>>> # Fully qualified
>>> doc.findtext('content/{http://www.w3.org/1999/xhtml}html/'
... ' {http://www.w3.org/1999/xhtml}head/{http://www.w3.org/1999/xhtml}title')
'Hello World'
>>>

```

You can often simplify matters for yourself by wrapping namespace handling up into a utility class.

```

class XMLNamespaces:
    def __init__(self, **kwargs):
        self.namespaces = {}
        for name, uri in kwargs.items():
            self.register(name, uri)
    def register(self, name, uri):
        self.namespaces[name] = '{'+uri+'}'
    def __call__(self, path):
        return path.format_map(self.namespaces)

```

To use this class, you do the following:

```

>>> ns = XMLNamespaces(html='http://www.w3.org/1999/xhtml')
>>> doc.find(ns('content/{html}html'))
<Element '{http://www.w3.org/1999/xhtml}html' at 0x1007767e0>
>>> doc.findtext(ns('content/{html}html/{html}head/{html}title'))
'Hello World'
>>>

```

## Discussion

Parsing XML documents that contain namespaces can be messy. The `XMLNamespaces` class is really just meant to clean it up slightly by allowing you to use the shortened namespace names in subsequent operations as opposed to fully qualified URIs.

Unfortunately, there is no mechanism in the basic `ElementTree` parser to get further information about namespaces. However, you can get a bit more information about the scope of namespace processing if you're willing to use the `iterparse()` function instead. For example:

```

>>> from xml.etree.ElementTree import iterparse
>>> for evt, elem in iterparse('ns2.xml', ('end', 'start-ns', 'end-ns')):
...     print(evt, elem)
...
end <Element 'author' at 0x10110de10>
start-ns ('', 'http://www.w3.org/1999/xhtml')
end <Element '{http://www.w3.org/1999/xhtml}title' at 0x1011131b0>
end <Element '{http://www.w3.org/1999/xhtml}head' at 0x1011130a8>
end <Element '{http://www.w3.org/1999/xhtml}h1' at 0x101113310>
end <Element '{http://www.w3.org/1999/xhtml}body' at 0x101113260>
end <Element '{http://www.w3.org/1999/xhtml}html' at 0x10110df70>
end-ns None
end <Element 'content' at 0x10110de68>
end <Element 'top' at 0x10110dd60>
>>> elem      # This is the topmost element
<Element 'top' at 0x10110dd60>
>>>

```

As a final note, if the text you are parsing makes use of namespaces in addition to other advanced XML features, you're really better off using the **lxml** library instead of `ElementTree`. For instance, **lxml** provides better support for validating documents against a DTD, more complete XPath support, and other advanced XML features. This recipe is really just a simple fix to make parsing a little easier.

## 6.8. Interacting with a Relational Database

### Problem

You need to select, insert, or delete rows in a relational database.

### Solution

A standard way to represent rows of data in Python is as a sequence of tuples. For example:

```

stocks = [
    ('GOOG', 100, 490.1),
    ('AAPL', 50, 545.75),
    ('FB', 150, 7.45),
    ('HPQ', 75, 33.2),
]

```

Given data in this form, it is relatively straightforward to interact with a relational database using Python's standard database API, as described in **PEP 249**. The gist of the API is that all operations on the database are carried out by SQL queries. Each row of input or output data is represented by a tuple.

To illustrate, you can use the `sqlite3` module that comes with Python. If you are using a different database (e.g., MySQL, Postgres, or ODBC), you'll have to install a third-party

module to support it. However, the underlying programming interface will be virtually the same, if not identical.

The first step is to connect to the database. Typically, you execute a `connect()` function, supplying parameters such as the name of the database, hostname, username, password, and other details as needed. For example:

```
>>> import sqlite3
>>> db = sqlite3.connect('database.db')
>>>
```

To do anything with the data, you next create a cursor. Once you have a cursor, you can start executing SQL queries. For example:

```
>>> c = db.cursor()
>>> c.execute('create table portfolio (symbol text, shares integer, price real)')
<sqlite3.Cursor object at 0x10067a730>
>>> db.commit()
>>>
```

To insert a sequence of rows into the data, use a statement like this:

```
>>> c.executemany('insert into portfolio values (?, ?, ?)', stocks)
<sqlite3.Cursor object at 0x10067a730>
>>> db.commit()
>>>
```

To perform a query, use a statement such as this:

```
>>> for row in db.execute('select * from portfolio'):
...     print(row)
...
('GOOG', 100, 490.1)
('AAPL', 50, 545.75)
('FB', 150, 7.45)
('HPQ', 75, 33.2)
>>>
```

If you want to perform queries that accept user-supplied input parameters, make sure you escape the parameters using `?` like this:

```
>>> min_price = 100
>>> for row in db.execute('select * from portfolio where price >= ?',
...                       (min_price,)):
...     print(row)
...
('GOOG', 100, 490.1)
('AAPL', 50, 545.75)
>>>
```

## Discussion

At a low level, interacting with a database is an extremely straightforward thing to do. You simply form SQL statements and feed them to the underlying module to either update the database or retrieve data. That said, there are still some tricky details you'll need to sort out on a case-by-case basis.

One complication is the mapping of data from the database into Python types. For entries such as dates, it is most common to use `datetime` instances from the `datetime` module, or possibly system timestamps, as used in the `time` module. For numerical data, especially financial data involving decimals, numbers may be represented as `Decimal` instances from the `decimal` module. Unfortunately, the exact mapping varies by database backend so you'll have to read the associated documentation.

Another extremely critical complication concerns the formation of SQL statement strings. You should never use Python string formatting operators (e.g., `%`) or the `.format()` method to create such strings. If the values provided to such formatting operators are derived from user input, this opens up your program to an SQL-injection attack (see <http://xkcd.com/327>). The special `?` wildcard in queries instructs the database backend to use its own string substitution mechanism, which (hopefully) will do it safely.

Sadly, there is some inconsistency across database backends with respect to the wildcard. Many modules use `?` or `%s`, while others may use a different symbol, such as `:0` or `:1`, to refer to parameters. Again, you'll have to consult the documentation for the database module you're using. The `paramstyle` attribute of a database module also contains information about the quoting style.

For simply pulling data in and out of a database table, using the database API is usually simple enough. If you're doing something more complicated, it may make sense to use a higher-level interface, such as that provided by an object-relational mapper. Libraries such as **SQLAlchemy** allow database tables to be described as Python classes and for database operations to be carried out while hiding most of the underlying SQL.

## 6.9. Decoding and Encoding Hexadecimal Digits

### Problem

You need to decode a string of hexadecimal digits into a byte string or encode a byte string as hex.

### Solution

If you simply need to decode or encode a raw string of hex digits, use the `binascii` module. For example:

```

>>> # Initial byte string
>>> s = b'hello'

>>> # Encode as hex
>>> import binascii
>>> h = binascii.b2a_hex(s)
>>> h
b'68656c6c6f'

>>> # Decode back to bytes
>>> binascii.a2b_hex(h)
b'hello'
>>>

```

Similar functionality can also be found in the `base64` module. For example:

```

>>> import base64
>>> h = base64.b16encode(s)
>>> h
b'68656c6c6f'
>>> base64.b16decode(h)
b'hello'
>>>

```

## Discussion

For the most part, converting to and from hex is straightforward using the functions shown. The main difference between the two techniques is in case folding. The `base64.b16decode()` and `base64.b16encode()` functions only operate with uppercase hexadecimal letters, whereas the functions in `binascii` work with either case.

It's also important to note that the output produced by the encoding functions is always a byte string. To coerce it to Unicode for output, you may need to add an extra decoding step. For example:

```

>>> h = base64.b16encode(s)
>>> print(h)
b'68656c6c6f'
>>> print(h.decode('ascii'))
68656c6c6f
>>>

```

When decoding hex digits, the `b16decode()` and `a2b_hex()` functions accept either bytes or unicode strings. However, those strings must only contain ASCII-encoded hexadecimal digits.



## 6.10. Decoding and Encoding Base64

### Problem

You need to decode or encode binary data using Base64 encoding.

### Solution

The `base64` module has two functions—`b64encode()` and `b64decode()`—that do exactly what you want. For example:

```
>>> # Some byte data
>>> s = b'hello'
>>> import base64

>>> # Encode as Base64
>>> a = base64.b64encode(s)
>>> a
b'aGVsbG8='

>>> # Decode from Base64
>>> base64.b64decode(a)
b'hello'
>>>
```

### Discussion

Base64 encoding is only meant to be used on byte-oriented data such as byte strings and byte arrays. Moreover, the output of the encoding process is always a byte string. If you are mixing Base64-encoded data with Unicode text, you may have to perform an extra decoding step. For example:

```
>>> a = base64.b64encode(s).decode('ascii')
>>> a
'aGVsbG8='
>>>
```

When decoding Base64, both byte strings and Unicode text strings can be supplied. However, Unicode strings can only contain ASCII characters.

## 6.11. Reading and Writing Binary Arrays of Structures

### Problem

You want to read or write data encoded as a binary array of uniform structures into Python tuples.

## Solution

To work with binary data, use the `struct` module. Here is an example of code that writes a list of Python tuples out to a binary file, encoding each tuple as a structure using `struct`:

```
from struct import Struct

def write_records(records, format, f):
    """
    Write a sequence of tuples to a binary file of structures.
    """
    record_struct = Struct(format)
    for r in records:
        f.write(record_struct.pack(*r))

# Example
if __name__ == '__main__':
    records = [ (1, 2.3, 4.5),
                (6, 7.8, 9.0),
                (12, 13.4, 56.7) ]

    with open('data.b', 'wb') as f:
        write_records(records, '<idd', f)
```

There are several approaches for reading this file back into a list of tuples. First, if you're going to read the file incrementally in chunks, you can write code such as this:

```
from struct import Struct

def read_records(format, f):
    record_struct = Struct(format)
    chunks = iter(lambda: f.read(record_struct.size), b'')
    return (record_struct.unpack(chunk) for chunk in chunks)

# Example
if __name__ == '__main__':
    with open('data.b', 'rb') as f:
        for rec in read_records('<idd', f):
            # Process rec
        ...
```

If you want to read the file entirely into a byte string with a single read and convert it piece by piece, you can write the following:

```
from struct import Struct

def unpack_records(format, data):
    record_struct = Struct(format)
    return (record_struct.unpack_from(data, offset)
            for offset in range(0, len(data), record_struct.size))
```

```

# Example
if __name__ == '__main__':
    with open('data.b', 'rb') as f:
        data = f.read()

    for rec in unpack_records('<idd', data):
        # Process rec
    ...

```

In both cases, the result is an iterable that produces the tuples originally stored when the file was created.

## Discussion

For programs that must encode and decode binary data, it is common to use the `struct` module. To declare a new structure, simply create an instance of `Struct` such as:

```

# Little endian 32-bit integer, two double precision floats
record_struct = Struct('<idd')

```

Structures are always defined using a set of structure codes such as `i`, `d`, `f`, and so forth [see [the Python documentation](#)]. These codes correspond to specific binary data types such as 32-bit integers, 64-bit floats, 32-bit floats, and so forth. The `<` in the first character specifies the byte ordering. In this example, it is indicating “little endian.” Change the character to `>` for big endian or `!` for network byte order.

The resulting `Struct` instance has various attributes and methods for manipulating structures of that type. The `size` attribute contains the size of the structure in bytes, which is useful to have in I/O operations. `pack()` and `unpack()` methods are used to pack and unpack data. For example:

```

>>> from struct import Struct
>>> record_struct = Struct('<idd')
>>> record_struct.size
20
>>> record_struct.pack(1, 2.0, 3.0)
b'\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x08@'
>>> record_struct.unpack(_)
(1, 2.0, 3.0)
>>>

```

Sometimes you’ll see the `pack()` and `unpack()` operations called as module-level functions, as in the following:

```

>>> import struct
>>> struct.pack('<idd', 1, 2.0, 3.0)
b'\x01\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x08@'
>>> struct.unpack('<idd', _)
(1, 2.0, 3.0)
>>>

```

This works, but feels less elegant than creating a single Struct instance—especially if the same structure appears in multiple places in your code. By creating a Struct instance, the format code is only specified once and all of the useful operations are grouped together nicely. This certainly makes it easier to maintain your code if you need to fiddle with the structure code (as you only have to change it in one place).

The code for reading binary structures involves a number of interesting, yet elegant programming idioms. In the `read_records()` function, `iter()` is being used to make an iterator that returns fixed-sized chunks. See [Recipe 5.8](#). This iterator repeatedly calls a user-supplied callable (e.g., `lambda: f.read(record_struct.size)`) until it returns a specified value (e.g., `b`), at which point iteration stops. For example:

```
>>> f = open('data.b', 'rb')
>>> chunks = iter(lambda: f.read(20), b'')
>>> chunks
<callable_iterator object at 0x10069e6d0>
>>> for chk in chunks:
...     print(chk)
...
b'\x01\x00\x00\x00ffffff\x02@\x00\x00\x00\x00\x00\x00\x12@'
b'\x06\x00\x00\x00333333\x1f@\x00\x00\x00\x00\x00\x00"@'
b'\x0c\x00\x00\x00\xcd\xcc\xcc\xcc\xcc*\x9a\x99\x99\x99YL@'
>>>
```

One reason for creating an iterable is that it nicely allows records to be created using a generator comprehension, as shown in the solution. If you didn't use this approach, the code might look like this:

```
def read_records(format, f):
    record_struct = Struct(format)
    while True:
        chk = f.read(record_struct.size)
        if chk == b'':
            break
        yield record_struct.unpack(chk)
    return records
```

In the `unpack_records()` function, a different approach using the `unpack_from()` method is used. `unpack_from()` is a useful method for extracting binary data from a larger binary array, because it does so without making any temporary objects or memory copies. You just give it a byte string (or any array) along with a byte offset, and it will unpack fields directly from that location.

If you used `unpack()` instead of `unpack_from()`, you would need to modify the code to make a lot of small slices and offset calculations. For example:

```
def unpack_records(format, data):
    record_struct = Struct(format)
    return (record_struct.unpack(data[offset:offset + record_struct.size])
            for offset in range(0, len(data), record_struct.size))
```

In addition to being more complicated to read, this version also requires a lot more work, as it performs various offset calculations, copies data, and makes small slice objects. If you're going to be unpacking a lot of structures from a large byte string you've already read, `unpack_from()` is a more elegant approach.

Unpacking records is one place where you might want to use `namedtuple` objects from the `collections` module. This allows you to set attribute names on the returned tuples. For example:

```
from collections import namedtuple

Record = namedtuple('Record', ['kind', 'x', 'y'])

with open('data.p', 'rb') as f:
    records = (Record(*r) for r in read_records('<idd', f))

for r in records:
    print(r.kind, r.x, r.y)
```

If you're writing a program that needs to work with a large amount of binary data, you may be better off using a library such as `numpy`. For example, instead of reading a binary into a list of tuples, you could read it into a structured array, like this:

```
>>> import numpy as np
>>> f = open('data.b', 'rb')
>>> records = np.fromfile(f, dtype='<i,<d,<d')
>>> records
array([(1, 2.3, 4.5), (6, 7.8, 9.0), (12, 13.4, 56.7)],
      dtype=[('f0', '<i4'), ('f1', '<f8'), ('f2', '<f8')])
>>> records[0]
(1, 2.3, 4.5)
>>> records[1]
(6, 7.8, 9.0)
>>>
```

Last, but not least, if you're faced with the task of reading binary data in some known file format (i.e., image formats, shape files, HDF5, etc.), check to see if a Python module already exists for it. There's no reason to reinvent the wheel if you don't have to.

## 6.12. Reading Nested and Variable-Sized Binary Structures

### Problem

You need to read complicated binary-encoded data that contains a collection of nested and/or variable-sized records. Such data might include images, video, shapefiles, and so on.

# Solution

The `struct` module can be used to decode and encode almost any kind of binary data structure. To illustrate the kind of data in question here, suppose you have this Python data structure representing a collection of points that make up a series of polygons:

```
polys = [
    [ (1.0, 2.5), (3.5, 4.0), (2.5, 1.5) ],
    [ (7.0, 1.2), (5.1, 3.0), (0.5, 7.5), (0.8, 9.0) ],
    [ (3.4, 6.3), (1.2, 0.5), (4.6, 9.2) ],
]
```

Now suppose this data was to be encoded into a binary file where the file started with the following header:

Byte	Type	Description
0	int	File code (0x1234, little endian)
4	double	Minimum x (little endian)
12	double	Minimum y (little endian)
20	double	Maximum x (little endian)
28	double	Maximum y (little endian)
36	int	Number of polygons (little endian)

Following the header, a series of polygon records follow, each encoded as follows:

Byte	Type	Description
0	int	Record length including length (N bytes)
4-N	Points	Pairs of (X,Y) coords as doubles

To write this file, you can use Python code like this:

```
import struct
import itertools

def write_polys(filename, polys):
    # Determine bounding box
    flattened = list(itertools.chain(*polys))
    min_x = min(x for x, y in flattened)
    max_x = max(x for x, y in flattened)
    min_y = min(y for x, y in flattened)
    max_y = max(y for x, y in flattened)

    with open(filename, 'wb') as f:
        f.write(struct.pack('<iddddi',
                           0x1234,
                           min_x, min_y,
                           max_x, max_y,
                           len(polys)))
```

```

for poly in polys:
    size = len(poly) * struct.calcsize('<dd')
    f.write(struct.pack('<i', size+4))
    for pt in poly:
        f.write(struct.pack('<dd', *pt))

# Call it with our polygon data
write_polys('polys.bin', polys)

```

To read the resulting data back, you can write very similar looking code using the `struct.unpack()` function, reversing the operations performed during writing. For example:

```

import struct

def read_polys(filename):
    with open(filename, 'rb') as f:
        # Read the header
        header = f.read(40)
        file_code, min_x, min_y, max_x, max_y, num_polys = \
            struct.unpack('<iddddi', header)

        polys = []
        for n in range(num_polys):
            pbytes, = struct.unpack('<i', f.read(4))
            poly = []
            for m in range(pbytes // 16):
                pt = struct.unpack('<dd', f.read(16))
                poly.append(pt)
            polys.append(poly)
    return polys

```

Although this code works, it's also a rather messy mix of small reads, struct unpacking, and other details. If code like this is used to process a real datafile, it can quickly become even messier. Thus, it's an obvious candidate for an alternative solution that might simplify some of the steps and free the programmer to focus on more important matters.

In the remainder of this recipe, a rather advanced solution for interpreting binary data will be built up in pieces. The goal will be to allow a programmer to provide a high-level specification of the file format, and to simply have the details of reading and unpacking all of the data worked out under the covers. As a forewarning, the code that follows may be the most advanced example in this entire book, utilizing various object-oriented programming and metaprogramming techniques. Be sure to carefully read the discussion section as well as cross-references to other recipes.

First, when reading binary data, it is common for the file to contain headers and other data structures. Although the `struct` module can unpack this data into a tuple, another way to represent such information is through the use of a class. Here's some code that allows just that:

```

import struct

class StructField:
    """
    Descriptor representing a simple structure field
    """
    def __init__(self, format, offset):
        self.format = format
        self.offset = offset
    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            r = struct.unpack_from(self.format,
                                   instance._buffer, self.offset)
            return r[0] if len(r) == 1 else r

class Structure:
    def __init__(self, bytedata):
        self._buffer = memoryview(bytedata)

```

This code uses a descriptor to represent each structure field. Each descriptor contains a struct-compatible format code along with a byte offset into an underlying memory buffer. In the `__get__()` method, the `struct.unpack_from()` function is used to unpack a value from the buffer without having to make extra slices or copies.

The `Structure` class just serves as a base class that accepts some byte data and stores it as the underlying memory buffer used by the `StructField` descriptor. The use of a `memoryview()` in this class serves a purpose that will become clear later.

Using this code, you can now define a structure as a high-level class that mirrors the information found in the tables that described the expected file format. For example:

```

class PolyHeader(Structure):
    file_code = StructField('<i', 0)
    min_x = StructField('<d', 4)
    min_y = StructField('<d', 12)
    max_x = StructField('<d', 20)
    max_y = StructField('<d', 28)
    num_polys = StructField('<i', 36)

```

Here is an example of using this class to read the header from the polygon data written earlier:

```

>>> f = open('polys.bin', 'rb')
>>> phead = PolyHeader(f.read(40))
>>> phead.file_code == 0x1234
True
>>> phead.min_x
0.5
>>> phead.min_y
0.5

```



```
>>> phead.max_x
7.0
>>> phead.max_y
9.2
>>> phead.num_polys
3
>>>
```

This is interesting, but there are a number of annoyances with this approach. For one, even though you get the convenience of a class-like interface, the code is rather verbose and requires the user to specify a lot of low-level detail (e.g., repeated uses of `StructField`, specification of offsets, etc.). The resulting class is also missing common conveniences such as providing a way to compute the total size of the structure.

Any time you are faced with class definitions that are overly verbose like this, you might consider the use of a class decorator or metaclass. One of the features of a metaclass is that it can be used to fill in a lot of low-level implementation details, taking that burden off of the user. As an example, consider this metaclass and slight reformulation of the `Structure` class:

```
class StructureMeta(type):
    """
    Metaclass that automatically creates StructField descriptors
    """
    def __init__(self, clsname, bases, clsdict):
        fields = getattr(self, '_fields_', [])
        byte_order = ''
        offset = 0
        for format, fieldname in fields:
            if format.startswith('<', '>', '!', '@'):
                byte_order = format[0]
                format = format[1:]
            format = byte_order + format
            setattr(self, fieldname, StructField(format, offset))
            offset += struct.calcsize(format)
        setattr(self, 'struct_size', offset)

class Structure(metaclass=StructureMeta):
    def __init__(self, bytedata):
        self._buffer = bytedata

    @classmethod
    def from_file(cls, f):
        return cls(f.read(cls.struct_size))
```

Using this new `Structure` class, you can now write a structure definition like this:

```
class PolyHeader(Structure):
    _fields_ = [
        ('<i', 'file_code'),
        ('d', 'min_x'),
        ('d', 'min_y'),
```

```

        ('d', 'max_x'),
        ('d', 'max_y'),
        ('i', 'num_polys')
    ]

```

As you can see, the specification is a lot less verbose. The added `from_file()` class method also makes it easier to read the data from a file without knowing any details about the size or structure of the data. For example:

```

>>> f = open('polys.bin', 'rb')
>>> phead = PolyHeader.from_file(f)
>>> phead.file_code == 0x1234
True
>>> phead.min_x
0.5
>>> phead.min_y
0.5
>>> phead.max_x
7.0
>>> phead.max_y
9.2
>>> phead.num_polys
3
>>>

```

Once you introduce a metaclass into the mix, you can build more intelligence into it. For example, suppose you want to support nested binary structures. Here's a reformulation of the metaclass along with a new supporting descriptor that allows it:

```

class NestedStruct:
    """
    Descriptor representing a nested structure
    """
    def __init__(self, name, struct_type, offset):
        self.name = name
        self.struct_type = struct_type
        self.offset = offset
    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            data = instance._buffer[self.offset:
                                     self.offset+self.struct_type.struct_size]
            result = self.struct_type(data)
            # Save resulting structure back on instance to avoid
            # further recomputation of this step
            setattr(instance, self.name, result)
            return result

class StructureMeta(type):
    """
    Metaclass that automatically creates StructField descriptors
    """

```

```

'''
def __init__(self, clsname, bases, clsdict):
    fields = getattr(self, '_fields_', [])
    byte_order = ''
    offset = 0
    for format, fieldname in fields:
        if isinstance(format, StructureMeta):
            setattr(self, fieldname,
                    NestedStruct(fieldname, format, offset))
            offset += format.struct_size
        else:
            if format.startswith('<', '>', '!', '@'):
                byte_order = format[0]
                format = format[1:]
            format = byte_order + format
            setattr(self, fieldname, StructField(format, offset))
            offset += struct.calcsize(format)
    setattr(self, 'struct_size', offset)

```

In this code, the `NestedStruct` descriptor is used to overlay another structure definition over a region of memory. It does this by taking a slice of the original memory buffer and using it to instantiate the given structure type. Since the underlying memory buffer was initialized as a `memoryview`, this slicing does not incur any extra memory copies. Instead, it's just an overlay on the original memory. Moreover, to avoid repeated instantiations, the descriptor then stores the resulting inner structure object on the instance using the same technique described in [Recipe 8.10](#).

Using this new formulation, you can start to write code like this:

```

class Point(Structure):
    _fields_ = [
        ('<d', 'x'),
        ('d', 'y')
    ]

class PolyHeader(Structure):
    _fields_ = [
        ('<i', 'file_code'),
        (Point, 'min'),           # nested struct
        (Point, 'max'),          # nested struct
        ('i', 'num_polys')
    ]

```

Amazingly, it will all still work as you expect. For example:

```

>>> f = open('polys.bin', 'rb')
>>> phead = PolyHeader.from_file(f)
>>> phead.file_code == 0x1234
True
>>> phead.min           # Nested structure
<__main__.Point object at 0x1006a48d0>
>>> phead.min.x

```

```

0.5
>>> phead.min.y
0.5
>>> phead.max.x
7.0
>>> phead.max.y
9.2
>>> phead.num_polys
3
>>>

```

At this point, a framework for dealing with fixed-sized records has been developed, but what about the variable-sized components? For example, the remainder of the polygon files contain sections of variable size.

One way to handle this is to write a class that simply represents a chunk of binary data along with a utility function for interpreting the contents in different ways. This is closely related to the code in [Recipe 6.11](#):

```

class SizedRecord:
    def __init__(self, bytedata):
        self._buffer = memoryview(bytedata)

    @classmethod
    def from_file(cls, f, size_fmt, includes_size=True):
        sz_nbytes = struct.calcsize(size_fmt)
        sz_bytes = f.read(sz_nbytes)
        sz, = struct.unpack(size_fmt, sz_bytes)
        buf = f.read(sz - includes_size * sz_nbytes)
        return cls(buf)

    def iter_as(self, code):
        if isinstance(code, str):
            s = struct.Struct(code)
            for off in range(0, len(self._buffer), s.size):
                yield s.unpack_from(self._buffer, off)
        elif isinstance(code, StructureMeta):
            size = code.struct_size
            for off in range(0, len(self._buffer), size):
                data = self._buffer[off:off+size]
                yield code(data)

```

The `SizedRecord.from_file()` class method is a utility for reading a size-prefixed chunk of data from a file, which is common in many file formats. As input, it accepts a structure format code containing the encoding of the size, which is expected to be in bytes. The optional `includes_size` argument specifies whether the number of bytes includes the size header or not. Here's an example of how you would use this code to read the individual polygons in the polygon file:

```

>>> f = open('polys.bin', 'rb')
>>> phead = PolyHeader.from_file(f)
>>> phead.num_polys
3
>>> polydata = [ SizedRecord.from_file(f, '<i')
...               for n in range(phead.num_polys) ]
>>> polydata
[<__main__.SizedRecord object at 0x1006a4d50>,
 <__main__.SizedRecord object at 0x1006a4f50>,
 <__main__.SizedRecord object at 0x10070da90>]
>>>

```

As shown, the contents of the `SizedRecord` instances have not yet been interpreted. To do that, use the `iter_as()` method, which accepts a structure format code or `Structure` class as input. This gives you a lot of flexibility in how to interpret the data. For example:

```

>>> for n, poly in enumerate(polydata):
...     print('Polygon', n)
...     for p in poly.iter_as('<dd'):
...         print(p)
...
Polygon 0
(1.0, 2.5)
(3.5, 4.0)
(2.5, 1.5)
Polygon 1
(7.0, 1.2)
(5.1, 3.0)
(0.5, 7.5)
(0.8, 9.0)
Polygon 2
(3.4, 6.3)
(1.2, 0.5)
(4.6, 9.2)
>>>

>>> for n, poly in enumerate(polydata):
...     print('Polygon', n)
...     for p in poly.iter_as(Point):
...         print(p.x, p.y)
...
Polygon 0
1.0 2.5
3.5 4.0
2.5 1.5
Polygon 1
7.0 1.2
5.1 3.0
0.5 7.5
0.8 9.0
Polygon 2

```

```
3.4 6.3
1.2 0.5
4.6 9.2
>>>
```

Putting all of this together, here's an alternative formulation of the `read_polys()` function:

```
class Point(Structure):
    _fields_ = [
        ('<d', 'x'),
        ('<d', 'y')
    ]

class PolyHeader(Structure):
    _fields_ = [
        ('<i', 'file_code'),
        (Point, 'min'),
        (Point, 'max'),
        ('i', 'num_polys')
    ]

def read_polys(filename):
    polys = []
    with open(filename, 'rb') as f:
        phead = PolyHeader.from_file(f)
        for n in range(phead.num_polys):
            rec = SizedRecord.from_file(f, '<i')
            poly = [ (p.x, p.y)
                     for p in rec.iter_as(Point) ]
            polys.append(poly)
    return polys
```

## Discussion

This recipe provides a practical application of various advanced programming techniques, including descriptors, lazy evaluation, metaclasses, class variables, and memoryviews. However, they all serve a very specific purpose.

A major feature of the implementation is that it is strongly based on the idea of lazy-unpacking. When an instance of `Structure` is created, the `__init__()` merely creates a memoryview of the supplied byte data and does nothing else. Specifically, no unpacking or other structure-related operations take place at this time. One motivation for taking this approach is that you might only be interested in a few specific parts of a binary record. Rather than unpacking the whole file, only the parts that are actually accessed will be unpacked.

To implement the lazy unpacking and packing of values, the `StructField` descriptor class is used. Each attribute the user lists in `_fields_` gets converted to a `StructField` descriptor that stores the associated structure format code and byte offset into

the stored buffer. The `StructureMeta` metaclass is what creates these descriptors automatically when various structure classes are defined. The main reason for using a metaclass is to make it extremely easy for a user to specify a structure format with a high-level description without worrying about low-level details.

One subtle aspect of the `StructureMeta` metaclass is that it makes byte order sticky. That is, if any attribute specifies a byte order (< for little endian or > for big endian), that ordering is applied to all fields that follow. This helps avoid extra typing, but also makes it possible to switch in the middle of a definition. For example, you might have something more complicated, such as this:

```
class ShapeFile(Structure):
    _fields_ = [ ('>i', 'file_code'),      # Big endian
                 ('20s', 'unused'),
                 ('i', 'file_length'),
                 ('<i', 'version'),       # Little endian
                 ('i', 'shape_type'),
                 ('d', 'min_x'),
                 ('d', 'min_y'),
                 ('d', 'max_x'),
                 ('d', 'max_y'),
                 ('d', 'min_z'),
                 ('d', 'max_z'),
                 ('d', 'min_m'),
                 ('d', 'max_m') ]
```

As noted, the use of a `memoryview()` in the solution serves a useful role in avoiding memory copies. When structures start to nest, `memoryviews` can be used to overlay different parts of the structure definition on the same region of memory. This aspect of the solution is subtle, but it concerns the slicing behavior of a `memoryview` versus a normal byte array. If you slice a byte string or byte array, you usually get a copy of the data. Not so with a `memoryview`—slices simply overlay the existing memory. Thus, this approach is more efficient.

A number of related recipes will help expand upon the topics used in the solution. See [Recipe 8.13](#) for a closely related recipe that uses descriptors to build a type system. [Recipe 8.10](#) has information about lazily computed properties and is related to the implementation of the `NestedStruct` descriptor. [Recipe 9.19](#) has an example of using a metaclass to initialize class members, much in the same manner as the `StructureMeta` class. The source code for Python's `ctypes` library may also be of interest, due to its similar support for defining data structures, nesting of data structures, and similar functionality.

## 6.13. Summarizing Data and Performing Statistics

### Problem

You need to crunch through large datasets and generate summaries or other kinds of statistics.

### Solution

For any kind of data analysis involving statistics, time series, and other related techniques, you should look at the [Pandas library](#).

To give you a taste, here's an example of using Pandas to analyze the City of Chicago [rat and rodent database](#). At the time of this writing, it's a CSV file with about 74,000 entries:

```
>>> import pandas

>>> # Read a CSV file, skipping last line
>>> rats = pandas.read_csv('rats.csv', skip_footer=1)
>>> rats
<class 'pandas.core.frame.DataFrame'>
Int64Index: 74055 entries, 0 to 74054
Data columns:
Creation Date          74055  non-null values
Status                74055  non-null values
Completion Date        72154  non-null values
Service Request Number 74055  non-null values
Type of Service Request 74055  non-null values
Number of Premises Baited 65804  non-null values
Number of Premises with Garbage 65600  non-null values
Number of Premises with Rats 65752  non-null values
Current Activity        66041  non-null values
Most Recent Action     66023  non-null values
Street Address         74055  non-null values
ZIP Code               73584  non-null values
X Coordinate           74043  non-null values
Y Coordinate           74043  non-null values
Ward                   74044  non-null values
Police District        74044  non-null values
Community Area         74044  non-null values
Latitude               74043  non-null values
Longitude              74043  non-null values
Location               74043  non-null values
dtypes: float64(11), object(9)

>>> # Investigate range of values for a certain field
>>> rats['Current Activity'].unique()
array([nan, Dispatch Crew, Request Sanitation Inspector], dtype=object)
```



```

>>> # Filter the data
>>> crew_dispatched = rats[rats['Current Activity'] == 'Dispatch Crew']
>>> len(crew_dispatched)
65676
>>>

>>> # Find 10 most rat-infested ZIP codes in Chicago
>>> crew_dispatched['ZIP Code'].value_counts()[:10]
60647    3837
60618    3530
60614    3284
60629    3251
60636    2801
60657    2465
60641    2238
60609    2206
60651    2152
60632    2071
>>>

>>> # Group by completion date
>>> dates = crew_dispatched.groupby('Completion Date')
<pandas.core.groupby.DataFrameGroupBy object at 0x10d0a2a10>
>>> len(dates)
472
>>>

>>> # Determine counts on each day
>>> date_counts = dates.size()
>>> date_counts[0:10]
Completion Date
01/03/2011      4
01/03/2012    125
01/04/2011     54
01/04/2012     38
01/05/2011     78
01/05/2012    100
01/06/2011    100
01/06/2012     58
01/07/2011      1
01/09/2012     12
>>>

>>> # Sort the counts
>>> date_counts.sort()
>>> date_counts[-10:]
Completion Date
10/12/2012    313
10/21/2011    314
09/20/2011    316
10/26/2011    319
02/22/2011    325

```

```
10/26/2012    333
03/17/2011    336
10/13/2011    378
10/14/2011    391
10/07/2011    457
>>>
```

Yes, October 7, 2011, was indeed a very busy day for rats.

## Discussion

Pandas is a large library that has more features than can be described here. However, if you need to analyze large datasets, group data, perform statistics, or other similar tasks, it's definitely worth a look.

*Python for Data Analysis* by Wes McKinney (O'Reilly) also contains much more information.

Defining functions using the `def` statement is a cornerstone of all programs. The goal of this chapter is to present some more advanced and unusual function definition and usage patterns. Topics include default arguments, functions that take any number of arguments, keyword-only arguments, annotations, and closures. In addition, some tricky control flow and data passing problems involving callback functions are addressed.

## 7.1. Writing Functions That Accept Any Number of Arguments

### Problem

You want to write a function that accepts any number of input arguments.

### Solution

To write a function that accepts any number of positional arguments, use a `*` argument. For example:

```
def avg(first, *rest):  
    return (first + sum(rest)) / (1 + len(rest))  
  
# Sample use  
avg(1, 2)           # 1.5  
avg(1, 2, 3, 4)     # 2.5
```

In this example, `rest` is a tuple of all the extra positional arguments passed. The code treats it as a sequence in performing subsequent calculations.

To accept any number of keyword arguments, use an argument that starts with \*\*. For example:

```
import html

def make_element(name, value, **attrs):
    keyvals = [' %s="%s"' % item for item in attrs.items()]
    attr_str = ''.join(keyvals)
    element = '<{name}{attrs}>{value}</{name}>'.format(
        name=name,
        attrs=attr_str,
        value=html.escape(value))
    return element

# Example
# Creates '<item size="large" quantity="6">Albatross</item>'
make_element('item', 'Albatross', size='large', quantity=6)

# Creates '<p>&lt;span&gt;</p>'
make_element('p', '<span>')
```

Here, `attrs` is a dictionary that holds the passed keyword arguments (if any).

If you want a function that can accept both any number of positional and keyword-only arguments, use `*` and `**` together. For example:

```
def anyargs(*args, **kwargs):
    print(args)      # A tuple
    print(kwargs)    # A dict
```

With this function, all of the positional arguments are placed into a tuple `args`, and all of the keyword arguments are placed into a dictionary `kwargs`.

## Discussion

A `*` argument can only appear as the last positional argument in a function definition. A `**` argument can only appear as the last argument. A subtle aspect of function definitions is that arguments can still appear after a `*` argument.

```
def a(x, *args, y):
    pass

def b(x, *args, y, **kwargs):
    pass
```

Such arguments are known as keyword-only arguments, and are discussed further in [Recipe 7.2](#).

## 7.2. Writing Functions That Only Accept Keyword Arguments

### Problem

You want a function to only accept certain arguments by keyword.

### Solution

This feature is easy to implement if you place the keyword arguments after a `*` argument or a single unnamed `*`. For example:

```
def recv(maxsize, *, block):
    'Receives a message'
    pass

recv(1024, True)          # TypeError
recv(1024, block=True)    # Ok
```

This technique can also be used to specify keyword arguments for functions that accept a varying number of positional arguments. For example:

```
def minimum(*values, clip=None):
    m = min(values)
    if clip is not None:
        m = clip if clip > m else m
    return m

minimum(1, 5, 2, -5, 10)    # Returns -5
minimum(1, 5, 2, -5, 10, clip=0) # Returns 0
```

### Discussion

Keyword-only arguments are often a good way to enforce greater code clarity when specifying optional function arguments. For example, consider a call like this:

```
msg = recv(1024, False)
```

If someone is not intimately familiar with the workings of the `recv()`, they may have no idea what the `False` argument means. On the other hand, it is much clearer if the call is written like this:

```
msg = recv(1024, block=False)
```

The use of keyword-only arguments is also often preferable to tricks involving `**kwargs`, since they show up properly when the user asks for help:

```
>>> help(recv)
Help on function recv in module __main__:
```

```
recv(maxsize, *, block)
    Receives a message
```

Keyword-only arguments also have utility in more advanced contexts. For example, they can be used to inject arguments into functions that make use of the `*args` and `**kwargs` convention for accepting all inputs. See [Recipe 9.11](#) for an example.

## 7.3. Attaching Informational Metadata to Function Arguments

### Problem

You’ve written a function, but would like to attach some additional information to the arguments so that others know more about how a function is supposed to be used.

### Solution

Function argument annotations can be a useful way to give programmers hints about how a function is supposed to be used. For example, consider the following annotated function:

```
def add(x:int, y:int) -> int:
    return x + y
```

The Python interpreter does not attach any semantic meaning to the attached annotations. They are not type checks, nor do they make Python behave any differently than it did before. However, they might give useful hints to others reading the source code about what you had in mind. Third-party tools and frameworks might also attach semantic meaning to the annotations. They also appear in documentation:

```
>>> help(add)
Help on function add in module __main__:

add(x: int, y: int) -> int
>>>
```

Although you can attach any kind of object to a function as an annotation (e.g., numbers, strings, instances, etc.), classes or strings often seem to make the most sense.

### Discussion

Function annotations are merely stored in a function’s `__annotations__` attribute. For example:

```
>>> add.__annotations__
{'y': <class 'int'>, 'return': <class 'int'>, 'x': <class 'int'>}
```

Although there are many potential uses of annotations, their primary utility is probably just documentation. Because Python doesn't have type declarations, it can often be difficult to know what you're supposed to pass into a function if you're simply reading its source code in isolation. An annotation gives someone more of a hint.

See [Recipe 9.20](#) for an advanced example showing how to use annotations to implement multiple dispatch (i.e., overloaded functions).

## 7.4. Returning Multiple Values from a Function

### Problem

You want to return multiple values from a function.

### Solution

To return multiple values from a function, simply return a tuple. For example:

```
>>> def myfun():
...     return 1, 2, 3
...
>>> a, b, c = myfun()
>>> a
1
>>> b
2
>>> c
3
```

### Discussion

Although it looks like `myfun()` returns multiple values, a tuple is actually being created. It looks a bit peculiar, but it's actually the comma that forms a tuple, not the parentheses. For example:

```
>>> a = (1, 2)      # With parentheses
>>> a
(1, 2)
>>> b = 1, 2        # Without parentheses
>>> b
(1, 2)
>>>
```

When calling functions that return a tuple, it is common to assign the result to multiple variables, as shown. This is simply tuple unpacking, as described in [Recipe 1.1](#). The return value could also have been assigned to a single variable:

```
>>> x = myfun()
>>> x
```

```
(1, 2, 3)
>>>
```

## 7.5. Defining Functions with Default Arguments

### Problem

You want to define a function or method where one or more of the arguments are optional and have a default value.

### Solution

On the surface, defining a function with optional arguments is easy—simply assign values in the definition and make sure that default arguments appear last. For example:

```
def spam(a, b=42):
    print(a, b)

spam(1)          # Ok. a=1, b=42
spam(1, 2)       # Ok. a=1, b=2
```

If the default value is supposed to be a mutable container, such as a list, set, or dictionary, use `None` as the default and write code like this:

```
# Using a list as a default value
def spam(a, b=None):
    if b is None:
        b = []
    ...
```

If, instead of providing a default value, you want to write code that merely tests whether an optional argument was given an interesting value or not, use this idiom:

```
_no_value = object()

def spam(a, b=_no_value):
    if b is _no_value:
        print('No b value supplied')
    ...
```

Here's how this function behaves:

```
>>> spam(1)
No b value supplied
>>> spam(1, 2)      # b = 2
>>> spam(1, None)   # b = None
>>>
```

Carefully observe that there is a distinction between passing no value at all and passing a value of `None`.



## Discussion

Defining functions with default arguments is easy, but there is a bit more to it than meets the eye.

First, the values assigned as a default are bound only once at the time of function definition. Try this example to see it:

```
>>> x = 42
>>> def spam(a, b=x):
...     print(a, b)
...
>>> spam(1)
1 42
>>> x = 23      # Has no effect
>>> spam(1)
1 42
>>>
```

Notice how changing the variable `x` (which was used as a default value) has no effect whatsoever. This is because the default value was fixed at function definition time.

Second, the values assigned as defaults should always be immutable objects, such as `None`, `True`, `False`, numbers, or strings. Specifically, never write code like this:

```
def spam(a, b=[]):    # NO!
...
...
```

If you do this, you can run into all sorts of trouble if the default value ever escapes the function and gets modified. Such changes will permanently alter the default value across future function calls. For example:

```
>>> def spam(a, b=[]):
...     print(b)
...     return b
...
>>> x = spam(1)
>>> x
[]
>>> x.append(99)
>>> x.append('Yow!')
>>> x
[99, 'Yow!']
>>> spam(1)      # Modified list gets returned!
[99, 'Yow!']
>>>
```

That's probably not what you want. To avoid this, it's better to assign `None` as a default and add a check inside the function for it, as shown in the solution.

The use of the `is` operator when testing for `None` is a critical part of this recipe. Sometimes people make this mistake:

```
def spam(a, b=None):
    if not b:          # NO! Use 'b is None' instead
        b = []
    ...
```

The problem here is that although `None` evaluates to `False`, many other objects (e.g., zero-length strings, lists, tuples, dicts, etc.) do as well. Thus, the test just shown would falsely treat certain inputs as missing. For example:

```
>>> spam(1)          # OK
>>> x = []
>>> spam(1, x)        # Silent error. x value overwritten by default
>>> spam(1, 0)        # Silent error. 0 ignored
>>> spam(1, '')       # Silent error. '' ignored
>>>
```

The last part of this recipe is something that’s rather subtle—a function that tests to see whether a value (any value) has been supplied to an optional argument or not. The tricky part here is that you can’t use a default value of `None`, `0`, or `False` to test for the presence of a user-supplied argument (since all of these are perfectly valid values that a user might supply). Thus, you need something else to test against.

To solve this problem, you can create a unique private instance of `object`, as shown in the solution (the `_no_value` variable). In the function, you then check the identity of the supplied argument against this special value to see if an argument was supplied or not. The thinking here is that it would be extremely unlikely for a user to pass the `_no_value` instance in as an input value. Therefore, it becomes a safe value to check against if you’re trying to determine whether an argument was supplied or not.

The use of `object()` might look rather unusual here. `object` is a class that serves as the common base class for almost all objects in Python. You can create instances of `object`, but they are wholly uninteresting, as they have no notable methods nor any instance data (because there is no underlying instance dictionary, you can’t even set any attributes). About the only thing you can do is perform tests for identity. This makes them useful as special values, as shown in the solution.

## 7.6. Defining Anonymous or Inline Functions

### Problem

You need to supply a short callback function for use with an operation such as `sort()`, but you don’t want to write a separate one-line function using the `def` statement. Instead, you’d like a shortcut that allows you to specify the function “in line.”

## Solution

Simple functions that do nothing more than evaluate an expression can be replaced by a `lambda` expression. For example:

```
>>> add = lambda x, y: x + y
>>> add(2,3)
5
>>> add('hello', 'world')
'helloworld'
>>>
```

The use of `lambda` here is the same as having typed this:

```
>>> def add(x, y):
...     return x + y
...
>>> add(2,3)
5
>>>
```

Typically, `lambda` is used in the context of some other operation, such as sorting or a data reduction:

```
>>> names = ['David Beazley', 'Brian Jones',
...          'Raymond Hettinger', 'Ned Batchelder']
>>> sorted(names, key=lambda name: name.split()[-1].lower())
['Ned Batchelder', 'David Beazley', 'Raymond Hettinger', 'Brian Jones']
>>>
```

## Discussion

Although `lambda` allows you to define a simple function, its use is highly restricted. In particular, only a single expression can be specified, the result of which is the return value. This means that no other language features, including multiple statements, conditionals, iteration, and exception handling, can be included.

You can quite happily write a lot of Python code without ever using `lambda`. However, you'll occasionally encounter it in programs where someone is writing a lot of tiny functions that evaluate various expressions, or in programs that require users to supply callback functions.

## 7.7. Capturing Variables in Anonymous Functions

### Problem

You've defined an anonymous function using `lambda`, but you also need to capture the values of certain variables at the time of definition.

## Solution

Consider the behavior of the following code:

```
>>> x = 10
>>> a = lambda y: x + y
>>> x = 20
>>> b = lambda y: x + y
>>>
```

Now ask yourself a question. What are the values of `a(10)` and `b(10)`? If you think the results might be 20 and 30, you would be wrong:

```
>>> a(10)
30
>>> b(10)
30
>>>
```

The problem here is that the value of `x` used in the `lambda` expression is a free variable that gets bound at runtime, not definition time. Thus, the value of `x` in the `lambda` expressions is whatever the value of the `x` variable happens to be at the time of execution. For example:

```
>>> x = 15
>>> a(10)
25
>>> x = 3
>>> a(10)
13
>>>
```

If you want an anonymous function to capture a value at the point of definition and keep it, include the value as a default value, like this:

```
>>> x = 10
>>> a = lambda y, x=x: x + y
>>> x = 20
>>> b = lambda y, x=x: x + y
>>> a(10)
20
>>> b(10)
30
>>>
```

## Discussion

The problem addressed in this recipe is something that tends to come up in code that tries to be just a little bit too clever with the use of `lambda` functions. For example, creating a list of `lambda` expressions using a list comprehension or in a loop of some

kind and expecting the lambda functions to remember the iteration variable at the time of definition. For example:

```
>>> funcs = [lambda x: x+n for n in range(5)]
>>> for f in funcs:
...     print(f(0))
...
4
4
4
4
4
>>>
```

Notice how all functions think that `n` has the last value during iteration. Now compare to the following:

```
>>> funcs = [lambda x, n=n: x+n for n in range(5)]
>>> for f in funcs:
...     print(f(0))
...
0
1
2
3
4
>>>
```

As you can see, the functions now capture the value of `n` at the time of definition.

## 7.8. Making an N-Argument Callable Work As a Callable with Fewer Arguments

### Problem

You have a callable that you would like to use with some other Python code, possibly as a callback function or handler, but it takes too many arguments and causes an exception when called.

### Solution

If you need to reduce the number of arguments to a function, you should use `functools.partial()`. The `partial()` function allows you to assign fixed values to one or more of the arguments, thus reducing the number of arguments that need to be supplied to subsequent calls. To illustrate, suppose you have this function:

```
def spam(a, b, c, d):
    print(a, b, c, d)
```

Now consider the use of `partial()` to fix certain argument values:

```
>>> from functools import partial
>>> s1 = partial(spam, 1)           # a = 1
>>> s1(2, 3, 4)
1 2 3 4
>>> s1(4, 5, 6)
1 4 5 6
>>> s2 = partial(spam, d=42)       # d = 42
>>> s2(1, 2, 3)
1 2 3 42
>>> s2(4, 5, 5)
4 5 5 42
>>> s3 = partial(spam, 1, 2, d=42) # a = 1, b = 2, d = 42
>>> s3(3)
1 2 3 42
>>> s3(4)
1 2 4 42
>>> s3(5)
1 2 5 42
>>>
```

Observe that `partial()` fixes the values for certain arguments and returns a new callable as a result. This new callable accepts the still unassigned arguments, combines them with the arguments given to `partial()`, and passes everything to the original function.

## Discussion

This recipe is really related to the problem of making seemingly incompatible bits of code work together. A series of examples will help illustrate.

As a first example, suppose you have a list of points represented as tuples of (x,y) coordinates. You could use the following function to compute the distance between two points:

```
points = [ (1, 2), (3, 4), (5, 6), (7, 8) ]

import math
def distance(p1, p2):
    x1, y1 = p1
    x2, y2 = p2
    return math.hypot(x2 - x1, y2 - y1)
```

Now suppose you want to sort all of the points according to their distance from some other point. The `sort()` method of lists accepts a key argument that can be used to customize sorting, but it only works with functions that take a single argument (thus, `distance()` is not suitable). Here's how you might use `partial()` to fix it:

```
>>> pt = (4, 3)
>>> points.sort(key=partial(distance,pt))
>>> points
[(3, 4), (1, 2), (5, 6), (7, 8)]
>>>
```

As an extension of this idea, `partial()` can often be used to tweak the argument signatures of callback functions used in other libraries. For example, here's a bit of code that uses `multiprocessing` to asynchronously compute a result which is handed to a callback function that accepts both the result and an optional logging argument:

```
def output_result(result, log=None):
    if log is not None:
        log.debug('Got: %r', result)

# A sample function
def add(x, y):
    return x + y

if __name__ == '__main__':
    import logging
    from multiprocessing import Pool
    from functools import partial

    logging.basicConfig(level=logging.DEBUG)
    log = logging.getLogger('test')

    p = Pool()
    p.apply_async(add, (3, 4), callback=partial(output_result, log=log))
    p.close()
    p.join()
```

When supplying the callback function using `apply_async()`, the extra logging argument is given using `partial()`. `multiprocessing` is none the wiser about all of this—it simply invokes the callback function with a single value.

As a similar example, consider the problem of writing network servers. The `socket` server module makes it relatively easy. For example, here is a simple echo server:

```
from socketserver import StreamRequestHandler, TCPServer

class EchoHandler(StreamRequestHandler):
    def handle(self):
        for line in self.rfile:
            self.wfile.write(b'GOT:' + line)

serv = TCPServer(('', 15000), EchoHandler)
serv.serve_forever()
```

However, suppose you want to give the `EchoHandler` class an `__init__()` method that accepts an additional configuration argument. For example:

```

class EchoHandler(StreamRequestHandler):
    # ack is added keyword-only argument. *args, **kwargs are
    # any normal parameters supplied (which are passed on)
    def __init__(self, *args, ack, **kwargs):
        self.ack = ack
        super().__init__(*args, **kwargs)
    def handle(self):
        for line in self.rfile:
            self.wfile.write(self.ack + line)

```

If you make this change, you'll find there is no longer an obvious way to plug it into the `TCPServer` class. In fact, you'll find that the code now starts generating exceptions like this:

```

Exception happened during processing of request from ('127.0.0.1', 59834)
Traceback (most recent call last):
...
TypeError: __init__() missing 1 required keyword-only argument: 'ack'

```

At first glance, it seems impossible to fix this code, short of modifying the source code to `socketserver` or coming up with some kind of weird workaround. However, it's easy to resolve using `partial()`—just use it to supply the value of the `ack` argument, like this:

```

from functools import partial
serv = TCPServer(('', 15000), partial(EchoHandler, ack=b'RECEIVED:'))
serv.serve_forever()

```

In this example, the specification of the `ack` argument in the `__init__()` method might look a little funny, but it's being specified as a keyword-only argument. This is discussed further in [Recipe 7.2](#).

The functionality of `partial()` is sometimes replaced with a `lambda` expression. For example, the previous examples might use statements such as this:

```

points.sort(key=lambda p: distance(pt, p))

p.apply_async(add, (3, 4), callback=lambda result: output_result(result, log))

serv = TCPServer(('', 15000),
                 lambda *args, **kwargs: EchoHandler(*args,
                                                         ack=b'RECEIVED:',
                                                         **kwargs))

```

This code works, but it's more verbose and potentially a lot more confusing to someone reading it. Using `partial()` is a bit more explicit about your intentions (supplying values for some of the arguments.)



## 7.9. Replacing Single Method Classes with Functions

### Problem

You have a class that only defines a single method besides `__init__()`. However, to simplify your code, you would much rather just have a simple function.

### Solution

In many cases, single-method classes can be turned into functions using closures. Consider, as an example, the following class, which allows a user to fetch URLs using a kind of templating scheme.

```
from urllib.request import urlopen

class UrlTemplate:
    def __init__(self, template):
        self.template = template
    def open(self, **kwargs):
        return urlopen(self.template.format_map(kwargs))

# Example use. Download stock data from yahoo
yahoo = UrlTemplate('http://finance.yahoo.com/d/quotes.csv?s={names}&f={fields}')
for line in yahoo.open(names='IBM,AAPL,FB', fields='sl1c1v'):
    print(line.decode('utf-8'))
```

The class could be replaced with a much simpler function:

```
def urltemplate(template):
    def opener(**kwargs):
        return urlopen(template.format_map(kwargs))
    return opener

# Example use
yahoo = urltemplate('http://finance.yahoo.com/d/quotes.csv?s={names}&f={fields}')
for line in yahoo(names='IBM,AAPL,FB', fields='sl1c1v'):
    print(line.decode('utf-8'))
```

### Discussion

In many cases, the only reason you might have a single-method class is to store additional state for use in the method. For example, the only purpose of the `UrlTemplate` class is to hold the `template` value someplace so that it can be used in the `open()` method.

Using an inner function or closure, as shown in the solution, is often more elegant. Simply stated, a closure is just a function, but with an extra environment of the variables that are used inside the function. A key feature of a closure is that it remembers the environment in which it was defined. Thus, in the solution, the `opener()` function remembers the value of the `template` argument, and uses it in subsequent calls.

Whenever you're writing code and you encounter the problem of attaching additional state to a function, think closures. They are often a more minimal and elegant solution than the alternative of turning your function into a full-fledged class.

## 7.10. Carrying Extra State with Callback Functions

### Problem

You're writing code that relies on the use of callback functions (e.g., event handlers, completion callbacks, etc.), but you want to have the callback function carry extra state for use inside the callback function.

### Solution

This recipe pertains to the use of callback functions that are found in many libraries and frameworks—especially those related to asynchronous processing. To illustrate and for the purposes of testing, define the following function, which invokes a callback:

```
def apply_async(func, args, *, callback):
    # Compute the result
    result = func(*args)

    # Invoke the callback with the result
    callback(result)
```

In reality, such code might do all sorts of advanced processing involving threads, processes, and timers, but that's not the main focus here. Instead, we're simply focused on the invocation of the callback. Here's an example that shows how the preceding code gets used:

```
>>> def print_result(result):
...     print('Got:', result)
...
>>> def add(x, y):
...     return x + y
...
>>> apply_async(add, (2, 3), callback=print_result)
Got: 5
>>> apply_async(add, ('hello', 'world'), callback=print_result)
Got: helloworld
>>>
```

As you will notice, the `print_result()` function only accepts a single argument, which is the result. No other information is passed in. This lack of information can sometimes present problems when you want the callback to interact with other variables or parts of the environment.

One way to carry extra information in a callback is to use a bound-method instead of a simple function. For example, this class keeps an internal sequence number that is incremented every time a result is received:

```
class ResultHandler:
    def __init__(self):
        self.sequence = 0
    def handler(self, result):
        self.sequence += 1
        print('[{}] Got: {}'.format(self.sequence, result))
```

To use this class, you would create an instance and use the bound method `handler` as the callback:

```
>>> r = ResultHandler()
>>> apply_async(add, (2, 3), callback=r.handler)
[1] Got: 5
>>> apply_async(add, ('hello', 'world'), callback=r.handler)
[2] Got: helloworld
>>>
```

As an alternative to a class, you can also use a closure to capture state. For example:

```
def make_handler():
    sequence = 0
    def handler(result):
        nonlocal sequence
        sequence += 1
        print('[{}] Got: {}'.format(sequence, result))
    return handler
```

Here is an example of this variant:

```
>>> handler = make_handler()
>>> apply_async(add, (2, 3), callback=handler)
[1] Got: 5
>>> apply_async(add, ('hello', 'world'), callback=handler)
[2] Got: helloworld
>>>
```

As yet another variation on this theme, you can sometimes use a coroutine to accomplish the same thing:

```
def make_handler():
    sequence = 0
    while True:
        result = yield
        sequence += 1
        print('[{}] Got: {}'.format(sequence, result))
```

For a coroutine, you would use its `send()` method as the callback, like this:

```
>>> handler = make_handler()
>>> next(handler)      # Advance to the yield
```

```
>>> apply_async(add, (2, 3), callback=handler.send)
[1] Got: 5
>>> apply_async(add, ('hello', 'world'), callback=handler.send)
[2] Got: helloworld
>>>
```

Last, but not least, you can also carry state into a callback using an extra argument and partial function application. For example:

```
>>> class SequenceNo:
...     def __init__(self):
...         self.sequence = 0
...
>>> def handler(result, seq):
...     seq.sequence += 1
...     print('[{}] Got: {}'.format(seq.sequence, result))
...
>>> seq = SequenceNo()
>>> from functools import partial
>>> apply_async(add, (2, 3), callback=partial(handler, seq=seq))
[1] Got: 5
>>> apply_async(add, ('hello', 'world'), callback=partial(handler, seq=seq))
[2] Got: helloworld
>>>
```

## Discussion

Software based on callback functions often runs the risk of turning into a huge tangled mess. Part of the issue is that the callback function is often disconnected from the code that made the initial request leading to callback execution. Thus, the execution environment between making the request and handling the result is effectively lost. If you want the callback function to continue with a procedure involving multiple steps, you have to figure out how to save and restore the associated state.

There are really two main approaches that are useful for capturing and carrying state. You can carry it around on an instance (attached to a bound method perhaps) or you can carry it around in a closure (an inner function). Of the two techniques, closures are perhaps a bit more lightweight and natural in that they are simply built from functions. They also automatically capture all of the variables being used. Thus, it frees you from having to worry about the exact state needs to be stored (it's determined automatically from your code).

If using closures, you need to pay careful attention to mutable variables. In the solution, the `nonlocal` declaration is used to indicate that the `sequence` variable is being modified from within the callback. Without this declaration, you'll get an error.

The use of a coroutine as a callback handler is interesting in that it is closely related to the closure approach. In some sense, it's even cleaner, since there is just a single function. Moreover, variables can be freely modified without worrying about `nonlocal` declara-

tions. The potential downside is that coroutines don't tend to be as well understood as other parts of Python. There are also a few tricky bits such as the need to call `next()` on a coroutine prior to using it. That's something that could be easy to forget in practice. Nevertheless, coroutines have other potential uses here, such as the definition of an inlined callback (covered in the next recipe).

The last technique involving `partial()` is useful if all you need to do is pass extra values into a callback. Instead of using `partial()`, you'll sometimes see the same thing accomplished with the use of a `lambda`:

```
>>> apply_async(add, (2, 3), callback=lambda r: handler(r, seq))
[1] Got: 5
>>>
```

For more examples, see [Recipe 7.8](#), which shows how to use `partial()` to change argument signatures.

## 7.11. Inlining Callback Functions

### Problem

You're writing code that uses callback functions, but you're concerned about the proliferation of small functions and mind boggling control flow. You would like some way to make the code look more like a normal sequence of procedural steps.

### Solution

Callback functions can be inlined into a function using generators and coroutines. To illustrate, suppose you have a function that performs work and invokes a callback as follows (see [Recipe 7.10](#)):

```
def apply_async(func, args, *, callback):
    # Compute the result
    result = func(*args)

    # Invoke the callback with the result
    callback(result)
```

Now take a look at the following supporting code, which involves an `Async` class and an `inlined_async` decorator:

```
from queue import Queue
from functools import wraps

class Async:
    def __init__(self, func, args):
        self.func = func
        self.args = args
```

```

def inlined_async(func):
    @wraps(func)
    def wrapper(*args):
        f = func(*args)
        result_queue = Queue()
        result_queue.put(None)
        while True:
            result = result_queue.get()
            try:
                a = f.send(result)
                apply_async(a.func, a.args, callback=result_queue.put)
            except StopIteration:
                break
        return wrapper

```

These two fragments of code will allow you to inline the callback steps using yield statements. For example:

```

def add(x, y):
    return x + y

@inlined_async
def test():
    r = yield Async(add, (2, 3))
    print(r)
    r = yield Async(add, ('hello', 'world'))
    print(r)
    for n in range(10):
        r = yield Async(add, (n, n))
        print(r)
    print('Goodbye')

```

If you call `test()`, you'll get output like this:

```

5
helloworld
0
2
4
6
8
10
12
14
16
18
Goodbye

```

Aside from the special decorator and use of `yield`, you will notice that no callback functions appear anywhere (except behind the scenes).

## Discussion

This recipe will really test your knowledge of callback functions, generators, and control flow.

First, in code involving callbacks, the whole point is that the current calculation will suspend and resume at some later point in time (e.g., asynchronously). When the calculation resumes, the callback will get executed to continue the processing. The `apply_async()` function illustrates the essential parts of executing the callback, although in reality it might be much more complicated (involving threads, processes, event handlers, etc.).

The idea that a calculation will suspend and resume naturally maps to the execution model of a generator function. Specifically, the `yield` operation makes a generator function emit a value and suspend. Subsequent calls to the `__next__()` or `send()` method of a generator will make it start again.

With this in mind, the core of this recipe is found in the `inline_async()` decorator function. The key idea is that the decorator will step the generator function through all of its `yield` statements, one at a time. To do this, a result queue is created and initially populated with a value of `None`. A loop is then initiated in which a result is popped off the queue and sent into the generator. This advances to the next `yield`, at which point an instance of `Async` is received. The loop then looks at the function and arguments, and initiates the asynchronous calculation `apply_async()`. However, the sneakiest part of this calculation is that instead of using a normal callback function, the callback is set to the queue `put()` method.

At this point, it is left somewhat open as to precisely what happens. The main loop immediately goes back to the top and simply executes a `get()` operation on the queue. If data is present, it must be the result placed there by the `put()` callback. If nothing is there, the operation blocks, waiting for a result to arrive at some future time. How that might happen depends on the precise implementation of the `apply_async()` function.

If you're doubtful that anything this crazy would work, you can try it with the multiprocessing library and have async operations executed in separate processes:

```
if __name__ == '__main__':
    import multiprocessing
    pool = multiprocessing.Pool()
    apply_async = pool.apply_async

    # Run the test function
    test()
```

Indeed, you'll find that it works, but unraveling the control flow might require more coffee.

Hiding tricky control flow behind generator functions is found elsewhere in the standard library and third-party packages. For example, the `@contextmanager` decorator in the `contextlib` performs a similar mind-bending trick that glues the entry and exit from a context manager together across a `yield` statement. The popular **Twisted** package has inlined callbacks that are also similar.

## 7.12. Accessing Variables Defined Inside a Closure

### Problem

You would like to extend a closure with functions that allow the inner variables to be accessed and modified.

### Solution

Normally, the inner variables of a closure are completely hidden to the outside world. However, you can provide access by writing accessor functions and attaching them to the closure as function attributes. For example:

```
def sample():
    n = 0
    # Closure function
    def func():
        print('n=', n)

    # Accessor methods for n
    def get_n():
        return n

    def set_n(value):
        nonlocal n
        n = value

    # Attach as function attributes
    func.get_n = get_n
    func.set_n = set_n
    return func
```

Here is an example of using this code:

```
>>> f = sample()
>>> f()
n= 0
>>> f.set_n(10)
>>> f()
n= 10
>>> f.get_n()
10
>>>
```



## Discussion

There are two main features that make this recipe work. First, `nonlocal` declarations make it possible to write functions that change inner variables. Second, function attributes allow the accessor methods to be attached to the closure function in a straightforward manner where they work a lot like instance methods (even though no class is involved).

A slight extension to this recipe can be made to have closures emulate instances of a class. All you need to do is copy the inner functions over to the dictionary of an instance and return it. For example:

```
import sys
class ClosureInstance:
    def __init__(self, locals=None):
        if locals is None:
            locals = sys._getframe(1).f_locals

        # Update instance dictionary with callables
        self.__dict__.update((key,value) for key, value in locals.items()
                             if callable(value) )

    # Redirect special methods
    def __len__(self):
        return self.__dict__['__len__']()

# Example use
def Stack():
    items = []

    def push(item):
        items.append(item)

    def pop():
        return items.pop()

    def __len__():
        return len(items)

    return ClosureInstance()
```

Here's an interactive session to show that it actually works:

```
>>> s = Stack()
>>> s
<__main__.ClosureInstance object at 0x10069ed10>
>>> s.push(10)
>>> s.push(20)
>>> s.push('Hello')
>>> len(s)
3
>>> s.pop()
'Hello'
```

```
>>> s.pop()
20
>>> s.pop()
10
>>>
```

Interestingly, this code runs a bit faster than using a normal class definition. For example, you might be inclined to test the performance against a class like this:

```
class Stack2:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def __len__(self):
        return len(self.items)
```

If you do, you'll get results similar to the following:

```
>>> from timeit import timeit
>>> # Test involving closures
>>> s = Stack()
>>> timeit('s.push(1);s.pop()', 'from __main__ import s')
0.9874754269840196
>>> # Test involving a class
>>> s = Stack2()
>>> timeit('s.push(1);s.pop()', 'from __main__ import s')
1.0707052160287276
>>>
```

As shown, the closure version runs about 8% faster. Most of that is coming from streamlined access to the instance variables. Closures are faster because there's no extra self variable involved.

Raymond Hettinger has devised an even more **diabolical variant of this idea**. However, should you be inclined to do something like this in your code, be aware that it's still a rather weird substitute for a real class. For example, major features such as inheritance, properties, descriptors, or class methods don't work. You also have to play some tricks to get special methods to work (e.g., see the implementation of `__len__()` in `ClosureInstance`).

Lastly, you'll run the risk of confusing people who read your code and wonder why it doesn't look anything like a normal class definition (of course, they'll also wonder why it's faster). Nevertheless, it's an interesting example of what can be done by providing access to the internals of a closure.

In the big picture, adding methods to closures might have more utility in settings where you want to do things like reset the internal state, flush buffers, clear caches, or have some kind of feedback mechanism.



---

# Classes and Objects

The primary focus of this chapter is to present recipes to common programming patterns related to class definitions. Topics include making objects support common Python features, usage of special methods, encapsulation techniques, inheritance, memory management, and useful design patterns.

## 8.1. Changing the String Representation of Instances

### Problem

You want to change the output produced by printing or viewing instances to something more sensible.

### Solution

To change the string representation of an instance, define the `__str__()` and `__repr__()` methods. For example:

```
class Pair:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __repr__(self):
        return 'Pair({0.x!r}, {0.y!r})'.format(self)
    def __str__(self):
        return '({0.x!s}, {0.y!s})'.format(self)
```

The `__repr__()` method returns the code representation of an instance, and is usually the text you would type to re-create the instance. The built-in `repr()` function returns this text, as does the interactive interpreter when inspecting values. The `__str__()` method converts the instance to a string, and is the output produced by the `str()` and `print()` functions. For example:

```

>>> p = Pair(3, 4)
>>> p
Pair(3, 4)          # __repr__() output
>>> print(p)
(3, 4)              # __str__() output
>>>

```

The implementation of this recipe also shows how different string representations may be used during formatting. Specifically, the special `!r` formatting code indicates that the output of `__repr__()` should be used instead of `__str__()`, the default. You can try this experiment with the preceding class to see this:

```

>>> p = Pair(3, 4)
>>> print('p is {0!r}'.format(p))
p is Pair(3, 4)
>>> print('p is {0}'.format(p))
p is (3, 4)
>>>

```

## Discussion

Defining `__repr__()` and `__str__()` is often good practice, as it can simplify debugging and instance output. For example, by merely printing or logging an instance, a programmer will be shown more useful information about the instance contents.

It is standard practice for the output of `__repr__()` to produce text such that `eval(repr(x)) == x`. If this is not possible or desired, then it is common to create a useful textual representation enclosed in `<` and `>` instead. For example:

```

>>> f = open('file.dat')
>>> f
<_io.TextIOWrapper name='file.dat' mode='r' encoding='UTF-8'>
>>>

```

If no `__str__()` is defined, the output of `__repr__()` is used as a fallback.

The use of `format()` in the solution might look a little funny, but the format code `{0.x}` specifies the `x`-attribute of argument 0. So, in the following function, the 0 is actually the instance `self`:

```

def __repr__(self):
    return 'Pair({0.x!r}, {0.y!r})'.format(self)

```

As an alternative to this implementation, you could also use the `%` operator and the following code:

```

def __repr__(self):
    return 'Pair(%r, %r)' % (self.x, self.y)

```

## 8.2. Customizing String Formatting

### Problem

You want an object to support customized formatting through the `format()` function and string method.

### Solution

To customize string formatting, define the `__format__()` method on a class. For example:

```
_formats = {
    'ymd' : '{d.year}-{d.month}-{d.day}',
    'mdy' : '{d.month}/{d.day}/{d.year}',
    'dmy' : '{d.day}/{d.month}/{d.year}'
}

class Date:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    def __format__(self, code):
        if code == '':
            code = 'ymd'
        fmt = _formats[code]
        return fmt.format(d=self)
```

Instances of the `Date` class now support formatting operations such as the following:

```
>>> d = Date(2012, 12, 21)
>>> format(d)
'2012-12-21'
>>> format(d, 'mdy')
'12/21/2012'
>>> 'The date is {:ymd}'.format(d)
'The date is 2012-12-21'
>>> 'The date is {:mdy}'.format(d)
'The date is 12/21/2012'
>>>
```

### Discussion

The `__format__()` method provides a hook into Python's string formatting functionality. It's important to emphasize that the interpretation of format codes is entirely up to the class itself. Thus, the codes can be almost anything at all. For example, consider the following from the `datetime` module:

```

>>> from datetime import date
>>> d = date(2012, 12, 21)
>>> format(d)
'2012-12-21'
>>> format(d, '%A, %B %d, %Y')
'Friday, December 21, 2012'
>>> 'The end is {:d %b %Y}. Goodbye'.format(d)
'The end is 21 Dec 2012. Goodbye'
>>>

```

There are some standard conventions for the formatting of the built-in types. See the [documentation for the string module](#) for a formal specification.

## 8.3. Making Objects Support the Context-Management Protocol

### Problem

You want to make your objects support the context-management protocol (the `with` statement).

### Solution

In order to make an object compatible with the `with` statement, you need to implement `__enter__()` and `__exit__()` methods. For example, consider the following class, which provides a network connection:

```

from socket import socket, AF_INET, SOCK_STREAM

class LazyConnection:
    def __init__(self, address, family=AF_INET, type=SOCK_STREAM):
        self.address = address
        self.family = AF_INET
        self.type = SOCK_STREAM
        self.sock = None

    def __enter__(self):
        if self.sock is not None:
            raise RuntimeError('Already connected')
        self.sock = socket(self.family, self.type)
        self.sock.connect(self.address)
        return self.sock

    def __exit__(self, exc_ty, exc_val, tb):
        self.sock.close()
        self.sock = None

```



The key feature of this class is that it represents a network connection, but it doesn't actually do anything initially (e.g., it doesn't establish a connection). Instead, the connection is established and closed using the `with` statement (essentially on demand). For example:

```
from functools import partial

conn = LazyConnection(('www.python.org', 80))
# Connection closed
with conn as s:
    # conn.__enter__() executes: connection open
    s.send(b'GET /index.html HTTP/1.0\r\n')
    s.send(b'Host: www.python.org\r\n')
    s.send(b'\r\n')
    resp = b''.join(iter(partial(s.recv, 8192), b''))
    # conn.__exit__() executes: connection closed
```

## Discussion

The main principle behind writing a context manager is that you're writing code that's meant to surround a block of statements as defined by the use of the `with` statement. When the `with` statement is first encountered, the `__enter__()` method is triggered. The return value of `__enter__()` (if any) is placed into the variable indicated with the `as` qualifier. Afterward, the statements in the body of the `with` statement execute. Finally, the `__exit__()` method is triggered to clean up.

This control flow happens regardless of what happens in the body of the `with` statement, including if there are exceptions. In fact, the three arguments to the `__exit__()` method contain the exception type, value, and traceback for pending exceptions (if any). The `__exit__()` method can choose to use the exception information in some way or to ignore it by doing nothing and returning `None` as a result. If `__exit__()` returns `True`, the exception is cleared as if nothing happened and the program continues executing statements immediately after the `with` block.

One subtle aspect of this recipe is whether or not the `LazyConnection` class allows nested use of the connection with multiple `with` statements. As shown, only a single socket connection at a time is allowed, and an exception is raised if a repeated `with` statement is attempted when a socket is already in use. You can work around this limitation with a slightly different implementation, as shown here:

```
from socket import socket, AF_INET, SOCK_STREAM

class LazyConnection:
    def __init__(self, address, family=AF_INET, type=SOCK_STREAM):
        self.address = address
        self.family = AF_INET
        self.type = SOCK_STREAM
        self.connections = []
```

```

def __enter__(self):
    sock = socket(self.family, self.type)
    sock.connect(self.address)
    self.connections.append(sock)
    return sock

def __exit__(self, exc_ty, exc_val, tb):
    self.connections.pop().close()

# Example use
from functools import partial

conn = LazyConnection(('www.python.org', 80))
with conn as s1:
    ...
    with conn as s2:
        ...
        # s1 and s2 are independent sockets

```

In this second version, the `LazyConnection` class serves as a kind of factory for connections. Internally, a list is used to keep a stack. Whenever `__enter__()` executes, it makes a new connection and adds it to the stack. The `__exit__()` method simply pops the last connection off the stack and closes it. It's subtle, but this allows multiple connections to be created at once with nested `with` statements, as shown.

Context managers are most commonly used in programs that need to manage resources such as files, network connections, and locks. A key part of such resources is they have to be explicitly closed or released to operate correctly. For instance, if you acquire a lock, then you have to make sure you release it, or else you risk deadlock. By implementing `__enter__()`, `__exit__()`, and using the `with` statement, it is much easier to avoid such problems, since the cleanup code in the `__exit__()` method is guaranteed to run no matter what.

An alternative formulation of context managers is found in the `contextmanager` module. See [Recipe 9.22](#). A thread-safe version of this recipe can be found in [Recipe 12.6](#).

## 8.4. Saving Memory When Creating a Large Number of Instances

### Problem

Your program creates a large number (e.g., millions) of instances and uses a large amount of memory.

## Solution

For classes that primarily serve as simple data structures, you can often greatly reduce the memory footprint of instances by adding the `__slots__` attribute to the class definition. For example:

```
class Date:
    __slots__ = ['year', 'month', 'day']
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day
```

When you define `__slots__`, Python uses a much more compact internal representation for instances. Instead of each instance consisting of a dictionary, instances are built around a small fixed-sized array, much like a tuple or list. Attribute names listed in the `__slots__` specifier are internally mapped to specific indices within this array. A side effect of using slots is that it is no longer possible to add new attributes to instances—you are restricted to only those attribute names listed in the `__slots__` specifier.

## Discussion

The memory saved by using slots varies according to the number and type of attributes stored. However, in general, the resulting memory use is comparable to that of storing data in a tuple. To give you an idea, storing a single `Date` instance without slots requires 428 bytes of memory on a 64-bit version of Python. If slots is defined, it drops to 156 bytes. In a program that manipulated a large number of dates all at once, this would make a significant reduction in overall memory use.

Although slots may seem like a feature that could be generally useful, you should resist the urge to use it in most code. There are many parts of Python that rely on the normal dictionary-based implementation. In addition, classes that define slots don't support certain features such as multiple inheritance. For the most part, you should only use slots on classes that are going to serve as frequently used data structures in your program (e.g., if your program created millions of instances of a particular class).

A common misperception of `__slots__` is that it is an encapsulation tool that prevents users from adding new attributes to instances. Although this is a side effect of using slots, this was never the original purpose. Instead, `__slots__` was always intended to be an optimization tool.

## 8.5. Encapsulating Names in a Class

### Problem

You want to encapsulate “private” data on instances of a class, but are concerned about Python’s lack of access control.

### Solution

Rather than relying on language features to encapsulate data, Python programmers are expected to observe certain naming conventions concerning the intended usage of data and methods. The first convention is that any name that starts with a single leading underscore (`_`) should always be assumed to be internal implementation. For example:

```
class A:
    def __init__(self):
        self._internal = 0    # An internal attribute
        self.public = 1      # A public attribute

    def public_method(self):
        """
        A public method
        """
        ...

    def _internal_method(self):
        ...
```

Python doesn’t actually prevent someone from accessing internal names. However, doing so is considered impolite, and may result in fragile code. It should be noted, too, that the use of the leading underscore is also used for module names and module-level functions. For example, if you ever see a module name that starts with a leading underscore (e.g., `_socket`), it’s internal implementation. Likewise, module-level functions such as `sys._getframe()` should only be used with great caution.

You may also encounter the use of two leading underscores (`__`) on names within class definitions. For example:

```
class B:
    def __init__(self):
        self.__private = 0
    def __private_method(self):
        ...
    def public_method(self):
        ...
        self.__private_method()
        ...
```

The use of double leading underscores causes the name to be mangled to something else. Specifically, the private attributes in the preceding class get renamed to `_B__private` and `_B__private_method`, respectively. At this point, you might ask what purpose such name mangling serves. The answer is inheritance—such attributes cannot be overridden via inheritance. For example:

```
class C(B):
    def __init__(self):
        super().__init__()
        self.__private = 1      # Does not override B.__private
    # Does not override B.__private_method()
    def __private_method(self):
        ...
```

Here, the private names `__private` and `__private_method` get renamed to `_C__private` and `_C__private_method`, which are different than the mangled names in the base class B.

## Discussion

The fact that there are two different conventions (single underscore versus double underscore) for “private” attributes leads to the obvious question of which style you should use. For most code, you should probably just make your nonpublic names start with a single underscore. If, however, you know that your code will involve subclassing, and there are internal attributes that should be hidden from subclasses, use the double underscore instead.

It should also be noted that sometimes you may want to define a variable that clashes with the name of a reserved word. For this, you should use a single trailing underscore. For example:

```
lambda_ = 2.0      # Trailing _ to avoid clash with lambda keyword
```

The reason for not using a leading underscore here is that it avoids confusion about the intended usage (i.e., the use of a leading underscore could be interpreted as a way to avoid a name collision rather than as an indication that the value is private). Using a single trailing underscore solves this problem.

## 8.6. Creating Managed Attributes

### Problem

You want to add extra processing (e.g., type checking or validation) to the getting or setting of an instance attribute.

## Solution

A simple way to customize access to an attribute is to define it as a “property.” For example, this code defines a property that adds simple type checking to an attribute:

```
class Person:
    def __init__(self, first_name):
        self.first_name = first_name

    # Getter function
    @property
    def first_name(self):
        return self._first_name

    # Setter function
    @first_name.setter
    def first_name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._first_name = value

    # Deleter function (optional)
    @first_name.deleter
    def first_name(self):
        raise AttributeError("Can't delete attribute")
```

In the preceding code, there are three related methods, all of which must have the same name. The first method is a getter function, and establishes `first_name` as being a property. The other two methods attach optional setter and deleter functions to the `first_name` property. It's important to stress that the `@first_name.setter` and `@first_name.deleter` decorators won't be defined unless `first_name` was already established as a property using `@property`.

A critical feature of a property is that it looks like a normal attribute, but access automatically triggers the getter, setter, and deleter methods. For example:

```
>>> a = Person('Guido')
>>> a.first_name          # Calls the getter
'Guido'
>>> a.first_name = 42     # Calls the setter
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "prop.py", line 14, in first_name
    raise TypeError('Expected a string')
TypeError: Expected a string
>>> del a.first_name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't delete attribute
>>>
```

When implementing a property, the underlying data (if any) still needs to be stored somewhere. Thus, in the get and set methods, you see direct manipulation of a `_first_name` attribute, which is where the actual data lives. In addition, you may ask why the `__init__()` method sets `self.first_name` instead of `self._first_name`. In this example, the entire point of the property is to apply type checking when setting an attribute. Thus, chances are you would also want such checking to take place during initialization. By setting `self.first_name`, the set operation uses the setter method (as opposed to bypassing it by accessing `self._first_name`).

Properties can also be defined for existing get and set methods. For example:

```
class Person:
    def __init__(self, first_name):
        self.set_first_name(first_name)

    # Getter function
    def get_first_name(self):
        return self._first_name

    # Setter function
    def set_first_name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._first_name = value

    # Deleter function (optional)
    def del_first_name(self):
        raise AttributeError("Can't delete attribute")

    # Make a property from existing get/set methods
    name = property(get_first_name, set_first_name, del_first_name)
```

## Discussion

A property attribute is actually a collection of methods bundled together. If you inspect a class with a property, you can find the raw methods in the `fget`, `fset`, and `fdel` attributes of the property itself. For example:

```
>>> Person.first_name.fget
<function Person.first_name at 0x1006a60e0>
>>> Person.first_name.fset
<function Person.first_name at 0x1006a6170>
>>> Person.first_name.fdel
<function Person.first_name at 0x1006a62e0>
>>>
```

Normally, you wouldn't call `fget` or `fset` directly, but they are triggered automatically when the property is accessed.

Properties should only be used in cases where you actually need to perform extra processing on attribute access. Sometimes programmers coming from languages such as Java feel that all access should be handled by getters and setters, and that they should write code like this:

```
class Person:
    def __init__(self, first_name):
        self.first_name = name
    @property
    def first_name(self):
        return self._first_name
    @first_name.setter
    def first_name(self, value):
        self._first_name = value
```

Don't write properties that don't actually add anything extra like this. For one, it makes your code more verbose and confusing to others. Second, it will make your program run a lot slower. Lastly, it offers no real design benefit. Specifically, if you later decide that extra processing needs to be added to the handling of an ordinary attribute, you could promote it to a property without changing existing code. This is because the syntax of code that accessed the attribute would remain unchanged.

Properties can also be a way to define computed attributes. These are attributes that are not actually stored, but computed on demand. For example:

```
import math
class Circle:
    def __init__(self, radius):
        self.radius = radius
    @property
    def area(self):
        return math.pi * self.radius ** 2
    @property
    def perimeter(self):
        return 2 * math.pi * self.radius
```

Here, the use of properties results in a very uniform instance interface in that `radius`, `area`, and `perimeter` are all accessed as simple attributes, as opposed to a mix of simple attributes and method calls. For example:

```
>>> c = Circle(4.0)
>>> c.radius
4.0
>>> c.area           # Notice lack of ()
50.26548245743669
>>> c.perimeter      # Notice lack of ()
25.132741228718345
>>>
```



Although properties give you an elegant programming interface, sometimes you actually may want to directly use getter and setter functions. For example:

```
>>> p = Person('Guido')
>>> p.get_first_name()
'Guido'
>>> p.set_first_name('Larry')
>>>
```

This often arises in situations where Python code is being integrated into a larger infrastructure of systems or programs. For example, perhaps a Python class is going to be plugged into a large distributed system based on remote procedure calls or distributed objects. In such a setting, it may be much easier to work with an explicit get/set method (as a normal method call) rather than a property that implicitly makes such calls.

Last, but not least, don't write Python code that features a lot of repetitive property definitions. For example:

```
class Person:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    @property
    def first_name(self):
        return self._first_name

    @first_name.setter
    def first_name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._first_name = value

    # Repeated property code, but for a different name (bad!)
    @property
    def last_name(self):
        return self._last_name

    @last_name.setter
    def last_name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._last_name = value
```

Code repetition leads to bloated, error prone, and ugly code. As it turns out, there are much better ways to achieve the same thing using descriptors or closures. See Recipes 8.9 and 9.21.

## 8.7. Calling a Method on a Parent Class

### Problem

You want to invoke a method in a parent class in place of a method that has been overridden in a subclass.

### Solution

To call a method in a parent (or superclass), use the `super()` function. For example:

```
class A:
    def spam(self):
        print('A.spam')

class B(A):
    def spam(self):
        print('B.spam')
        super().spam()    # Call parent spam()
```

A very common use of `super()` is in the handling of the `__init__()` method to make sure that parents are properly initialized:

```
class A:
    def __init__(self):
        self.x = 0

class B(A):
    def __init__(self):
        super().__init__()
        self.y = 1
```

Another common use of `super()` is in code that overrides any of Python's special methods. For example:

```
class Proxy:
    def __init__(self, obj):
        self._obj = obj

    # Delegate attribute lookup to internal obj
    def __getattr__(self, name):
        return getattr(self._obj, name)

    # Delegate attribute assignment
    def __setattr__(self, name, value):
        if name.startswith('_'):
            super().__setattr__(name, value)    # Call original __setattr__
        else:
            setattr(self._obj, name, value)
```

In this code, the implementation of `__setattr__()` includes a name check. If the name starts with an underscore (`_`), it invokes the original implementation of `__setattr__()` using `super()`. Otherwise, it delegates to the internally held object `self._obj`. It looks a little funny, but `super()` works even though there is no explicit base class listed.

## Discussion

Correct use of the `super()` function is actually one of the most poorly understood aspects of Python. Occasionally, you will see code written that directly calls a method in a parent like this:

```
class Base:
    def __init__(self):
        print('Base.__init__')

class A(Base):
    def __init__(self):
        Base.__init__(self)
        print('A.__init__')
```

Although this “works” for most code, it can lead to bizarre trouble in advanced code involving multiple inheritance. For example, consider the following:

```
class Base:
    def __init__(self):
        print('Base.__init__')

class A(Base):
    def __init__(self):
        Base.__init__(self)
        print('A.__init__')

class B(Base):
    def __init__(self):
        Base.__init__(self)
        print('B.__init__')

class C(A,B):
    def __init__(self):
        A.__init__(self)
        B.__init__(self)
        print('C.__init__')
```

If you run this code, you’ll see that the `Base.__init__()` method gets invoked twice, as shown here:

```
>>> c = C()
Base.__init__
A.__init__
Base.__init__
```

```

B.__init__
C.__init__
>>>

```

Perhaps double-invocation of `Base.__init__()` is harmless, but perhaps not. If, on the other hand, you change the code to use `super()`, it all works:

```

class Base:
    def __init__(self):
        print('Base.__init__')

class A(Base):
    def __init__(self):
        super().__init__()
        print('A.__init__')

class B(Base):
    def __init__(self):
        super().__init__()
        print('B.__init__')

class C(A,B):
    def __init__(self):
        super().__init__()    # Only one call to super() here
        print('C.__init__')

```

When you use this new version, you'll find that each `__init__()` method only gets called once:

```

>>> c = C()
Base.__init__
B.__init__
A.__init__
C.__init__
>>>

```

To understand why it works, we need to step back for a minute and discuss how Python implements inheritance. For every class that you define, Python computes what's known as a method resolution order (MRO) list. The MRO list is simply a linear ordering of all the base classes. For example:

```

>>> C.__mro__
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>,
 <class '__main__.Base'>, <class 'object'>)
>>>

```

To implement inheritance, Python starts with the leftmost class and works its way left-to-right through classes on the MRO list until it finds the first attribute match.

The actual determination of the MRO list itself is made using a technique known as C3 Linearization. Without getting too bogged down in the mathematics of it, it is actually a merge sort of the MROs from the parent classes subject to three constraints:

- Child classes get checked before parents
- Multiple parents get checked in the order listed.
- If there are two valid choices for the next class, pick the one from the first parent.

Honestly, all you really need to know is that the order of classes in the MRO list “makes sense” for almost any class hierarchy you are going to define.

When you use the `super()` function, Python continues its search starting with the next class on the MRO. As long as every redefined method consistently uses `super()` and only calls it once, control will ultimately work its way through the entire MRO list and each method will only be called once. This is why you don’t get double calls to `Base.__init__()` in the second example.

A somewhat surprising aspect of `super()` is that it doesn’t necessarily go to the direct parent of a class next in the MRO and that you can even use it in a class with no direct parent at all. For example, consider this class:

```
class A:
    def spam(self):
        print('A.spam')
        super().spam()
```

If you try to use this class, you’ll find that it’s completely broken:

```
>>> a = A()
>>> a.spam()
A.spam
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "<stdin>", line 4, in spam
AttributeError: 'super' object has no attribute 'spam'
>>>
```

Yet, watch what happens if you start using the class with multiple inheritance:

```
>>> class B:
...     def spam(self):
...         print('B.spam')
...
>>> class C(A,B):
...     pass
...
>>> c = C()
>>> c.spam()
A.spam
B.spam
>>>
```

Here you see that the use of `super().spam()` in class A has, in fact, called the `spam()` method in class B—a class that is completely unrelated to A! This is all explained by the MRO of class C:

```
>>> C.__mro__
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>,
 <class 'object'>)
>>>
```

Using `super()` in this manner is most common when defining mixin classes. See Recipes 8.13 and 8.18.

However, because `super()` might invoke a method that you're not expecting, there are a few general rules of thumb you should try to follow. First, make sure that all methods with the same name in an inheritance hierarchy have a compatible calling signature (i.e., same number of arguments, argument names). This ensures that `super()` won't get tripped up if it tries to invoke a method on a class that's not a direct parent. Second, it's usually a good idea to make sure that the topmost class provides an implementation of the method so that the chain of lookups that occur along the MRO get terminated by an actual method of some sort.

Use of `super()` is sometimes a source of debate in the Python community. However, all things being equal, you should probably use it in modern code. Raymond Hettinger has written an excellent blog post “[Python's super\(\) Considered Super!](#)” that has even more examples and reasons why `super()` might be super-awesome.

## 8.8. Extending a Property in a Subclass

### Problem

Within a subclass, you want to extend the functionality of a property defined in a parent class.

### Solution

Consider the following code, which defines a property:

```
class Person:
    def __init__(self, name):
        self.name = name

    # Getter function
    @property
    def name(self):
        return self._name

    # Setter function
    @name.setter
```

```

def name(self, value):
    if not isinstance(value, str):
        raise TypeError('Expected a string')
    self._name = value

# Deleter function
@name.deleter
def name(self):
    raise AttributeError("Can't delete attribute")

```

Here is an example of a class that inherits from `Person` and extends the `name` property with new functionality:

```

class SubPerson(Person):
    @property
    def name(self):
        print('Getting name')
        return super().name

    @name.setter
    def name(self, value):
        print('Setting name to', value)
        super(SubPerson, SubPerson).name.__set__(self, value)

    @name.deleter
    def name(self):
        print('Deleting name')
        super(SubPerson, SubPerson).name.__delete__(self)

```

Here is an example of the new class in use:

```

>>> s = SubPerson('Guido')
Setting name to Guido
>>> s.name
Getting name
'Guido'
>>> s.name = 'Larry'
Setting name to Larry
>>> s.name = 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example.py", line 16, in name
    raise TypeError('Expected a string')
TypeError: Expected a string
>>>

```

If you only want to extend one of the methods of a property, use code such as the following:

```

class SubPerson(Person):
    @Person.name.getter
    def name(self):
        print('Getting name')
        return super().name

```

Or, alternatively, for just the setter, use this code:

```
class SubPerson(Person):
    @Person.name.setter
    def name(self, value):
        print('Setting name to', value)
        super(SubPerson, SubPerson).name.__set__(self, value)
```

## Discussion

Extending a property in a subclass introduces a number of very subtle problems related to the fact that a property is defined as a collection of getter, setter, and deleter methods, as opposed to just a single method. Thus, when extending a property, you need to figure out if you will redefine all of the methods together or just one of the methods.

In the first example, all of the property methods are redefined together. Within each method, `super()` is used to call the previous implementation. The use of `super(SubPerson, SubPerson).name.__set__(self, value)` in the setter function is no mistake. To delegate to the previous implementation of the setter, control needs to pass through the `__set__()` method of the previously defined `name` property. However, the only way to get to this method is to access it as a class variable instead of an instance variable. This is what happens with the `super(SubPerson, SubPerson)` operation.

If you only want to redefine one of the methods, it's not enough to use `@property` by itself. For example, code like this doesn't work:

```
class SubPerson(Person):
    @property
    def name(self):
        print('Getting name')
        return super().name
```

If you try the resulting code, you'll find that the setter function disappears entirely:

```
>>> s = SubPerson('Guido')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    s.name = 'Guido'
  File "example.py", line 5, in __init__
    self.name = name
AttributeError: can't set attribute
>>>
```

Instead, you should change the code to that shown in the solution:

```
class SubPerson(Person):
    @Person.name.getter
    def name(self):
        print('Getting name')
        return super().name
```



When you do this, all of the previously defined methods of the property are copied, and the getter function is replaced. It now works as expected:

```
>>> s = SubPerson('Guido')
>>> s.name
Getting name
'Guido'
>>> s.name = 'Larry'
>>> s.name
Getting name
'Larry'
>>> s.name = 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example.py", line 16, in name
    raise TypeError('Expected a string')
TypeError: Expected a string
>>>
```

In this particular solution, there is no way to replace the hardcoded class name `Person` with something more generic. If you don't know which base class defined a property, you should use the solution where all of the property methods are redefined and `super()` is used to pass control to the previous implementation.

It's worth noting that the first technique shown in this recipe can also be used to extend a descriptor, as described in [Recipe 8.9](#). For example:

```
# A descriptor
class String:
    def __init__(self, name):
        self.name = name

    def __get__(self, instance, cls):
        if instance is None:
            return self
        return instance.__dict__[self.name]

    def __set__(self, instance, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        instance.__dict__[self.name] = value

# A class with a descriptor
class Person:
    name = String('name')
    def __init__(self, name):
        self.name = name

# Extending a descriptor with a property
class SubPerson(Person):
    @property
    def name(self):
```

```

    print('Getting name')
    return super().name

@name.setter
def name(self, value):
    print('Setting name to', value)
    super(SubPerson, SubPerson).name.__set__(self, value)

@name.deleter
def name(self):
    print('Deleting name')
    super(SubPerson, SubPerson).name.__delete__(self)

```

Finally, it's worth noting that by the time you read this, subclassing of setter and deleter methods might be somewhat simplified. The solution shown will still work, but the bug reported at [Python's issues page](#) might resolve into a cleaner approach in a future Python version.

## 8.9. Creating a New Kind of Class or Instance Attribute

### Problem

You want to create a new kind of instance attribute type with some extra functionality, such as type checking.

### Solution

If you want to create an entirely new kind of instance attribute, define its functionality in the form of a descriptor class. Here is an example:

```

# Descriptor attribute for an integer type-checked attribute
class Integer:
    def __init__(self, name):
        self.name = name

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return instance.__dict__[self.name]

    def __set__(self, instance, value):
        if not isinstance(value, int):
            raise TypeError('Expected an int')
        instance.__dict__[self.name] = value

    def __delete__(self, instance):
        del instance.__dict__[self.name]

```

A descriptor is a class that implements the three core attribute access operations (get, set, and delete) in the form of `__get__()`, `__set__()`, and `__delete__()` special methods. These methods work by receiving an instance as input. The underlying dictionary of the instance is then manipulated as appropriate.

To use a descriptor, instances of the descriptor are placed into a class definition as class variables. For example:

```
class Point:
    x = Integer('x')
    y = Integer('y')
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

When you do this, all access to the descriptor attributes (e.g., `x` or `y`) is captured by the `__get__()`, `__set__()`, and `__delete__()` methods. For example:

```
>>> p = Point(2, 3)
>>> p.x           # Calls Point.x.__get__(p, Point)
2
>>> p.y = 5       # Calls Point.y.__set__(p, 5)
>>> p.x = 2.3     # Calls Point.x.__set__(p, 2.3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "descrip.py", line 12, in __set__
    raise TypeError('Expected an int')
TypeError: Expected an int
>>>
```

As input, each method of a descriptor receives the instance being manipulated. To carry out the requested operation, the underlying instance dictionary (the `__dict__` attribute) is manipulated as appropriate. The `self.name` attribute of the descriptor holds the dictionary key being used to store the actual data in the instance dictionary.

## Discussion

Descriptors provide the underlying magic for most of Python's class features, including `@classmethod`, `@staticmethod`, `@property`, and even the `__slots__` specification.

By defining a descriptor, you can capture the core instance operations (get, set, delete) at a very low level and completely customize what they do. This gives you great power, and is one of the most important tools employed by the writers of advanced libraries and frameworks.

One confusion with descriptors is that they can only be defined at the class level, not on a per-instance basis. Thus, code like this will not work:

```
# Does NOT work
class Point:
```

```
def __init__(self, x, y):
    self.x = Integer('x')    # No! Must be a class variable
    self.y = Integer('y')
    self.x = x
    self.y = y
```

Also, the implementation of the `__get__()` method is trickier than it seems:

```
# Descriptor attribute for an integer type-checked attribute
class Integer:
    ...
    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return instance.__dict__[self.name]
    ...
```

The reason `__get__()` looks somewhat complicated is to account for the distinction between instance variables and class variables. If a descriptor is accessed as a class variable, the `instance` argument is set to `None`. In this case, it is standard practice to simply return the descriptor instance itself (although any kind of custom processing is also allowed). For example:

```
>>> p = Point(2,3)
>>> p.x    # Calls Point.x.__get__(p, Point)
2
>>> Point.x # Calls Point.x.__get__(None, Point)
<__main__.Integer object at 0x100671890>
>>>
```

Descriptors are often just one component of a larger programming framework involving decorators or metaclasses. As such, their use may be hidden just barely out of sight. As an example, here is some more advanced descriptor-based code involving a class decorator:

```
# Descriptor for a type-checked attribute
class Typed:
    def __init__(self, name, expected_type):
        self.name = name
        self.expected_type = expected_type

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return instance.__dict__[self.name]

    def __set__(self, instance, value):
        if not isinstance(value, self.expected_type):
            raise TypeError('Expected ' + str(self.expected_type))
        instance.__dict__[self.name] = value
```

```

def __delete__(self, instance):
    del instance.__dict__[self.name]

# Class decorator that applies it to selected attributes
def typeassert(**kwargs):
    def decorate(cls):
        for name, expected_type in kwargs.items():
            # Attach a Typed descriptor to the class
            setattr(cls, name, Typed(name, expected_type))
        return cls
    return decorate

# Example use
@typeassert(name=str, shares=int, price=float)
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

```

Finally, it should be stressed that you would probably not write a descriptor if you simply want to customize the access of a single attribute of a specific class. For that, it's easier to use a property instead, as described in [Recipe 8.6](#). Descriptors are more useful in situations where there will be a lot of code reuse (i.e., you want to use the functionality provided by the descriptor in hundreds of places in your code or provide it as a library feature).

## 8.10. Using Lazily Computed Properties

### Problem

You'd like to define a read-only attribute as a property that only gets computed on access. However, once accessed, you'd like the value to be cached and not recomputed on each access.

### Solution

An efficient way to define a lazy attribute is through the use of a descriptor class, such as the following:

```

class lazyproperty:
    def __init__(self, func):
        self.func = func

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:

```

```

        value = self.func(instance)
        setattr(instance, self.func.__name__, value)
    return value

```

To utilize this code, you would use it in a class such as the following:

```

import math

class Circle:
    def __init__(self, radius):
        self.radius = radius

    @lazyproperty
    def area(self):
        print('Computing area')
        return math.pi * self.radius ** 2

    @lazyproperty
    def perimeter(self):
        print('Computing perimeter')
        return 2 * math.pi * self.radius

```

Here is an interactive session that illustrates how it works:

```

>>> c = Circle(4.0)
>>> c.radius
4.0
>>> c.area
Computing area
50.26548245743669
>>> c.area
50.26548245743669
>>> c.perimeter
Computing perimeter
25.132741228718345
>>> c.perimeter
25.132741228718345
>>>

```

Carefully observe that the messages “Computing area” and “Computing perimeter” only appear once.

## Discussion

In many cases, the whole point of having a lazily computed attribute is to improve performance. For example, you avoid computing values unless you actually need them somewhere. The solution shown does just this, but it exploits a subtle feature of descriptors to do it in a highly efficient way.

As shown in other recipes (e.g., [Recipe 8.9](#)), when a descriptor is placed into a class definition, its `__get__()`, `__set__()`, and `__delete__()` methods get triggered on attribute access. However, if a descriptor only defines a `__get__()` method, it has a much

weaker binding than usual. In particular, the `__get__()` method only fires if the attribute being accessed is not in the underlying instance dictionary.

The `lazyproperty` class exploits this by having the `__get__()` method store the computed value on the instance using the same name as the property itself. By doing this, the value gets stored in the instance dictionary and disables further computation of the property. You can observe this by digging a little deeper into the example:

```
>>> c = Circle(4.0)
>>> # Get instance variables
>>> vars(c)
{'radius': 4.0}

>>> # Compute area and observe variables afterward
>>> c.area
Computing area
50.26548245743669
>>> vars(c)
{'area': 50.26548245743669, 'radius': 4.0}

>>> # Notice access doesn't invoke property anymore
>>> c.area
50.26548245743669

>>> # Delete the variable and see property trigger again
>>> del c.area
>>> vars(c)
{'radius': 4.0}
>>> c.area
Computing area
50.26548245743669
>>>
```

One possible downside to this recipe is that the computed value becomes mutable after it's created. For example:

```
>>> c.area
Computing area
50.26548245743669
>>> c.area = 25
>>> c.area
25
>>>
```

If that's a concern, you can use a slightly less efficient implementation, like this:

```
def lazyproperty(func):
    name = '_lazy_' + func.__name__
    @property
    def lazy(self):
        if hasattr(self, name):
            return getattr(self, name)
        else:
```

```

        value = func(self)
        setattr(self, name, value)
        return value
    return lazy

```

If you use this version, you'll find that set operations are not allowed. For example:

```

>>> c = Circle(4.0)
>>> c.area
Computing area
50.26548245743669
>>> c.area
50.26548245743669
>>> c.area = 25
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>

```

However, a disadvantage is that all get operations have to be routed through the property's getter function. This is less efficient than simply looking up the value in the instance dictionary, as was done in the original solution.

For more information on properties and managed attributes, see [Recipe 8.6](#). Descriptors are described in [Recipe 8.9](#).

## 8.11. Simplifying the Initialization of Data Structures

### Problem

You are writing a lot of classes that serve as data structures, but you are getting tired of writing highly repetitive and boilerplate `__init__()` functions.

### Solution

You can often generalize the initialization of data structures into a single `__init__()` function defined in a common base class. For example:

```

class Structure:
    # Class variable that specifies expected fields
    _fields= []
    def __init__(self, *args):
        if len(args) != len(self._fields):
            raise TypeError('Expected {} arguments'.format(len(self._fields)))

    # Set the arguments
    for name, value in zip(self._fields, args):
        setattr(self, name, value)

```



```

# Example class definitions
if __name__ == '__main__':
    class Stock(Structure):
        _fields = ['name', 'shares', 'price']

    class Point(Structure):
        _fields = ['x', 'y']

    class Circle(Structure):
        _fields = ['radius']
        def area(self):
            return math.pi * self.radius ** 2

```

If you use the resulting classes, you'll find that they are easy to construct. For example:

```

>>> s = Stock('ACME', 50, 91.1)
>>> p = Point(2, 3)
>>> c = Circle(4.5)
>>> s2 = Stock('ACME', 50)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "structure.py", line 6, in __init__
    raise TypeError('Expected {} arguments'.format(len(self._fields)))
TypeError: Expected 3 arguments

```

Should you decide to support keyword arguments, there are several design options. One choice is to map the keyword arguments so that they only correspond to the attribute names specified in `_fields`. For example:

```

class Structure:
    _fields= []
    def __init__(self, *args, **kwargs):
        if len(args) > len(self._fields):
            raise TypeError('Expected {} arguments'.format(len(self._fields)))

        # Set all of the positional arguments
        for name, value in zip(self._fields, args):
            setattr(self, name, value)

        # Set the remaining keyword arguments
        for name in self._fields[len(args):]:
            setattr(self, name, kwargs.pop(name))

        # Check for any remaining unknown arguments
        if kwargs:
            raise TypeError('Invalid argument(s): {}'.format(','.join(kwargs)))

# Example use
if __name__ == '__main__':
    class Stock(Structure):
        _fields = ['name', 'shares', 'price']

    s1 = Stock('ACME', 50, 91.1)

```

```
s2 = Stock('ACME', 50, price=91.1)
s3 = Stock('ACME', shares=50, price=91.1)
```

Another possible choice is to use keyword arguments as a means for adding additional attributes to the structure not specified in `_fields`. For example:

```
class Structure:
    # Class variable that specifies expected fields
    _fields= []
    def __init__(self, *args, **kwargs):
        if len(args) != len(self._fields):
            raise TypeError('Expected {} arguments'.format(len(self._fields)))

        # Set the arguments
        for name, value in zip(self._fields, args):
            setattr(self, name, value)

        # Set the additional arguments (if any)
        extra_args = kwargs.keys() - self._fields
        for name in extra_args:
            setattr(self, name, kwargs.pop(name))
        if kwargs:
            raise TypeError('Duplicate values for {}'.format(','.join(kwargs)))

# Example use
if __name__ == '__main__':
    class Stock(Structure):
        _fields = ['name', 'shares', 'price']

    s1 = Stock('ACME', 50, 91.1)
    s2 = Stock('ACME', 50, 91.1, date='8/2/2012')
```

## Discussion

This technique of defining a general purpose `__init__()` method can be extremely useful if you're ever writing a program built around a large number of small data structures. It leads to much less code than manually writing `__init__()` methods like this:

```
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

class Circle:
    def __init__(self, radius):
        self.radius = radius
```

```
def area(self):
    return math.pi * self.radius ** 2
```

One subtle aspect of the implementation concerns the mechanism used to set value using the `setattr()` function. Instead of doing that, you might be inclined to directly access the instance dictionary. For example:

```
class Structure:
    # Class variable that specifies expected fields
    _fields= []
    def __init__(self, *args):
        if len(args) != len(self._fields):
            raise TypeError('Expected {} arguments'.format(len(self._fields)))

    # Set the arguments (alternate)
    self.__dict__.update(zip(self._fields,args))
```

Although this works, it's often not safe to make assumptions about the implementation of a subclass. If a subclass decided to use `__slots__` or wrap a specific attribute with a property (or descriptor), directly accessing the instance dictionary would break. The solution has been written to be as general purpose as possible and not to make any assumptions about subclasses.

A potential downside of this technique is that it impacts documentation and help features of IDEs. If a user asks for help on a specific class, the required arguments aren't described in the usual way. For example:

```
>>> help(Stock)
Help on class Stock in module __main__:

class Stock(Structure)
...
|   Methods inherited from Structure:
|   __init__(self, *args, **kwargs)
|
...
>>>
```

Many of these problems can be fixed by either attaching or enforcing a type signature in the `__init__()` function. See [Recipe 9.16](#).

It should be noted that it is also possible to automatically initialize instance variables using a utility function and a so-called “frame hack.” For example:

```
def init_fromlocals(self):
    import sys
    locs = sys._getframe(1).f_locals
    for k, v in locs.items():
        if k != 'self':
            setattr(self, k, v)
```

```
class Stock:
    def __init__(self, name, shares, price):
        init_fromlocals(self)
```

In this variation, the `init_fromlocals()` function uses `sys._getframe()` to peek at the local variables of the calling method. If used as the first step of an `__init__()` method, the local variables will be the same as the passed arguments and can be easily used to set attributes with the same names. Although this approach avoids the problem of getting the right calling signature in IDEs, it runs more than 50% slower than the solution provided in the recipe, requires more typing, and involves more sophisticated magic behind the scenes. If your code doesn't need this extra power, often times the simpler solution will work just fine.

## 8.12. Defining an Interface or Abstract Base Class

### Problem

You want to define a class that serves as an interface or abstract base class from which you can perform type checking and ensure that certain methods are implemented in subclasses.

### Solution

To define an abstract base class, use the `abc` module. For example:

```
from abc import ABCMeta, abstractmethod

class IStream(metaclass=ABCMeta):
    @abstractmethod
    def read(self, maxbytes=-1):
        pass
    @abstractmethod
    def write(self, data):
        pass
```

A central feature of an abstract base class is that it cannot be instantiated directly. For example, if you try to do it, you'll get an error:

```
a = IStream()  # TypeError: Can't instantiate abstract class
               # IStream with abstract methods read, write
```

Instead, an abstract base class is meant to be used as a base class for other classes that are expected to implement the required methods. For example:

```
class SocketStream(IStream):
    def read(self, maxbytes=-1):
        ...
    def write(self, data):
        ...
```

A major use of abstract base classes is in code that wants to enforce an expected programming interface. For example, one way to view the `IStream` base class is as a high-level specification for an interface that allows reading and writing of data. Code that explicitly checks for this interface could be written as follows:

```
def serialize(obj, stream):
    if not isinstance(stream, IStream):
        raise TypeError('Expected an IStream')
    ...
```

You might think that this kind of type checking only works by subclassing the abstract base class (ABC), but ABCs allow other classes to be registered as implementing the required interface. For example, you can do this:

```
import io

# Register the built-in I/O classes as supporting our interface
IStream.register(io.IOBase)

# Open a normal file and type check
f = open('foo.txt')
isinstance(f, IStream)      # Returns True
```

It should be noted that `@abstractmethod` can also be applied to static methods, class methods, and properties. You just need to make sure you apply it in the proper sequence where `@abstractmethod` appears immediately before the function definition, as shown here:

```
from abc import ABCMeta, abstractmethod

class A(metaclass=ABCMeta):
    @property
    @abstractmethod
    def name(self):
        pass

    @name.setter
    @abstractmethod
    def name(self, value):
        pass

    @classmethod
    @abstractmethod
    def method1(cls):
        pass

    @staticmethod
    @abstractmethod
    def method2():
        pass
```

## Discussion

Predefined abstract base classes are found in various places in the standard library. The `collections` module defines a variety of ABCs related to containers and iterators (sequences, mappings, sets, etc.), the `numbers` library defines ABCs related to numeric objects (integers, floats, rationals, etc.), and the `io` library defines ABCs related to I/O handling.

You can use the predefined ABCs to perform more generalized kinds of type checking. Here are some examples:

```
import collections

# Check if x is a sequence
if isinstance(x, collections.Sequence):
    ...

# Check if x is iterable
if isinstance(x, collections.Iterable):
    ...

# Check if x has a size
if isinstance(x, collections.Sized):
    ...

# Check if x is a mapping
if isinstance(x, collections.Mapping):
    ...
```

It should be noted that, as of this writing, certain library modules don't make use of these predefined ABCs as you might expect. For example:

```
from decimal import Decimal
import numbers

x = Decimal('3.4')
isinstance(x, numbers.Real)  # Returns False
```

Even though the value `3.4` is technically a real number, it doesn't type check that way to help avoid inadvertent mixing of floating-point numbers and decimals. Thus, if you use the ABC functionality, it is wise to carefully write tests that verify that the behavior is as you intended.

Although ABCs facilitate type checking, it's not something that you should overuse in a program. At its heart, Python is a dynamic language that gives you great flexibility. Trying to enforce type constraints everywhere tends to result in code that is more complicated than it needs to be. You should embrace Python's flexibility.

## 8.13. Implementing a Data Model or Type System

### Problem

You want to define various kinds of data structures, but want to enforce constraints on the values that are allowed to be assigned to certain attributes.

### Solution

In this problem, you are basically faced with the task of placing checks or assertions on the setting of certain instance attributes. To do this, you need to customize the setting of attributes on a per-attribute basis. To do this, you should use descriptors.

The following code illustrates the use of descriptors to implement a system type and value checking framework:

```
# Base class. Uses a descriptor to set a value
class Descriptor:
    def __init__(self, name=None, **opts):
        self.name = name
        for key, value in opts.items():
            setattr(self, key, value)

    def __set__(self, instance, value):
        instance.__dict__[self.name] = value

# Descriptor for enforcing types
class Typed(Descriptor):
    expected_type = type(None)

    def __set__(self, instance, value):
        if not isinstance(value, self.expected_type):
            raise TypeError('expected ' + str(self.expected_type))
        super().__set__(instance, value)

# Descriptor for enforcing values
class Unsigned(Descriptor):
    def __set__(self, instance, value):
        if value < 0:
            raise ValueError('Expected >= 0')
        super().__set__(instance, value)

class MaxSized(Descriptor):
    def __init__(self, name=None, **opts):
        if 'size' not in opts:
            raise TypeError('missing size option')
        super().__init__(name, **opts)

    def __set__(self, instance, value):
        if len(value) >= self.size:
```

```

        raise ValueError('size must be < ' + str(self.size))
    super().__set__(instance, value)

```

These classes should be viewed as basic building blocks from which you construct a data model or type system. Continuing, here is some code that implements some different kinds of data:

```

class Integer(Typed):
    expected_type = int

class UnsignedInteger(Integer, Unsigned):
    pass

class Float(Typed):
    expected_type = float

class UnsignedFloat(Float, Unsigned):
    pass

class String(Typed):
    expected_type = str

class SizedString(String, MaxSized):
    pass

```

Using these type objects, it is now possible to define a class such as this:

```

class Stock:
    # Specify constraints
    name = SizedString('name',size=8)
    shares = UnsignedInteger('shares')
    price = UnsignedFloat('price')
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

```

With the constraints in place, you'll find that assigning of attributes is now validated. For example:

```

>>> s = Stock('ACME', 50, 91.1)
>>> s.name
'ACME'
>>> s.shares = 75
>>> s.shares = -10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example.py", line 17, in __set__
    super().__set__(instance, value)
  File "example.py", line 23, in __set__
    raise ValueError('Expected >= 0')
ValueError: Expected >= 0
>>> s.price = 'a lot'

```



```

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example.py", line 16, in __set__
    raise TypeError('expected ' + str(self.expected_type))
TypeError: expected <class 'float'>
>>> s.name = 'ABRACADABRA'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example.py", line 17, in __set__
    super().__set__(instance, value)
  File "example.py", line 35, in __set__
    raise ValueError('size must be < ' + str(self.size))
ValueError: size must be < 8
>>>

```

There are some techniques that can be used to simplify the specification of constraints in classes. One approach is to use a class decorator, like this:

```

# Class decorator to apply constraints
def check_attributes(**kwargs):
    def decorate(cls):
        for key, value in kwargs.items():
            if isinstance(value, Descriptor):
                value.name = key
                setattr(cls, key, value)
            else:
                setattr(cls, key, value(key))
        return cls
    return decorate

# Example
@check_attributes(name=SizedString(size=8),
                  shares=UnsignedInteger,
                  price=UnsignedFloat)
class Stock:
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

```

Another approach to simplify the specification of constraints is to use a metaclass. For example:

```

# A metaclass that applies checking
class checkedmeta(type):
    def __new__(cls, clsname, bases, methods):
        # Attach attribute names to the descriptors
        for key, value in methods.items():
            if isinstance(value, Descriptor):
                value.name = key
        return type.__new__(cls, clsname, bases, methods)

```

```
# Example
class Stock(metaclass=checkedmeta):
    name = SizedString(size=8)
    shares = UnsignedInteger()
    price = UnsignedFloat()
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price
```

## Discussion

This recipe involves a number of advanced techniques, including descriptors, mixin classes, the use of `super()`, class decorators, and metaclasses. Covering the basics of all those topics is beyond what can be covered here, but examples can be found in other recipes (see Recipes 8.9, 8.18, 9.12, and 9.19). However, there are a number of subtle points worth noting.

First, in the `Descriptor` base class, you will notice that there is a `__set__()` method, but no corresponding `__get__()`. If a descriptor will do nothing more than extract an identically named value from the underlying instance dictionary, defining `__get__()` is unnecessary. In fact, defining `__get__()` will just make it run slower. Thus, this recipe only focuses on the implementation of `__set__()`.

The overall design of the various descriptor classes is based on mixin classes. For example, the `Unsigned` and `MaxSized` classes are meant to be mixed with the other descriptor classes derived from `Typed`. To handle a specific kind of data type, multiple inheritance is used to combine the desired functionality.

You will also notice that all `__init__()` methods of the various descriptors have been programmed to have an identical signature involving keyword arguments `**opts`. The class for `MaxSized` looks for its required attribute in `opts`, but simply passes it along to the `Descriptor` base class, which actually sets it. One tricky part about composing classes like this (especially mixins), is that you don't always know how the classes are going to be chained together or what `super()` will invoke. For this reason, you need to make it work with any possible combination of classes.

The definitions of the various type classes such as `Integer`, `Float`, and `String` illustrate a useful technique of using class variables to customize an implementation. The `Typed` descriptor merely looks for an `expected_type` attribute that is provided by each of those subclasses.

The use of a class decorator or metaclass is often useful for simplifying the specification by the user. You will notice that in those examples, the user no longer has to type the name of the attribute more than once. For example:

```

# Normal
class Point:
    x = Integer('x')
    y = Integer('y')

# Metaclass
class Point(metaclass=checkedmeta):
    x = Integer()
    y = Integer()

```

The code for the class decorator and metaclass simply scan the class dictionary looking for descriptors. When found, they simply fill in the descriptor name based on the key value.

Of all the approaches, the class decorator solution may provide the most flexibility and sanity. For one, it does not rely on any advanced machinery, such as metaclasses. Second, decoration is something that can easily be added or removed from a class definition as desired. For example, within the decorator, there could be an option to simply omit the added checking altogether. These might allow the checking to be something that could be turned on or off depending on demand (maybe for debugging versus production).

As a final twist, a class decorator approach can also be used as a replacement for mixin classes, multiple inheritance, and tricky use of the `super()` function. Here is an alternative formulation of this recipe that uses class decorators:

```

# Base class. Uses a descriptor to set a value
class Descriptor:
    def __init__(self, name=None, **opts):
        self.name = name
        for key, value in opts.items():
            setattr(self, key, value)

    def __set__(self, instance, value):
        instance.__dict__[self.name] = value

# Decorator for applying type checking
def Typed(expected_type, cls=None):
    if cls is None:
        return lambda cls: Typed(expected_type, cls)

    super_set = cls.__set__
    def __set__(self, instance, value):
        if not isinstance(value, expected_type):
            raise TypeError('expected ' + str(expected_type))
        super_set(self, instance, value)
    cls.__set__ = __set__
    return cls

# Decorator for unsigned values
def Unsigned(cls):
    super_set = cls.__set__

```

```

def __set__(self, instance, value):
    if value < 0:
        raise ValueError('Expected >= 0')
    super_set(self, instance, value)
cls.__set__ = __set__
return cls

# Decorator for allowing sized values
def MaxSized(cls):
    super_init = cls.__init__
    def __init__(self, name=None, **opts):
        if 'size' not in opts:
            raise TypeError('missing size option')
        super_init(self, name, **opts)
    cls.__init__ = __init__

    super_set = cls.__set__
    def __set__(self, instance, value):
        if len(value) >= self.size:
            raise ValueError('size must be < ' + str(self.size))
        super_set(self, instance, value)
    cls.__set__ = __set__
    return cls

# Specialized descriptors
@Typed(int)
class Integer(Descriptor):
    pass

@Unsigned
class UnsignedInteger(Integer):
    pass

@Typed(float)
class Float(Descriptor):
    pass

@Unsigned
class UnsignedFloat(Float):
    pass

@Typed(str)
class String(Descriptor):
    pass

@MaxSized
class SizedString(String):
    pass

```

The classes defined in this alternative formulation work in exactly the same manner as before (none of the earlier example code changes) except that everything runs much faster. For example, a simple timing test of setting a typed attribute reveals that the class

decorator approach runs almost 100% faster than the approach using mixins. Now aren't you glad you read all the way to the end?

## 8.14. Implementing Custom Containers

### Problem

You want to implement a custom class that mimics the behavior of a common built-in container type, such as a list or dictionary. However, you're not entirely sure what methods need to be implemented to do it.

### Solution

The `collections` library defines a variety of abstract base classes that are extremely useful when implementing custom container classes. To illustrate, suppose you want your class to support iteration. To do that, simply start by having it inherit from `collections.Iterable`, as follows:

```
import collections

class A(collections.Iterable):
    pass
```

The special feature about inheriting from `collections.Iterable` is that it ensures you implement all of the required special methods. If you don't, you'll get an error upon instantiation:

```
>>> a = A()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class A with abstract methods __iter__
>>>
```

To fix this error, simply give the class the required `__iter__()` method and implement it as desired (see Recipes 4.2 and 4.7).

Other notable classes defined in `collections` include `Sequence`, `MutableSequence`, `Mapping`, `MutableMapping`, `Set`, and `MutableSet`. Many of these classes form hierarchies with increasing levels of functionality (e.g., one such hierarchy is `Container`, `Iterable`, `Sized`, `Sequence`, and `MutableSequence`). Again, simply instantiate any of these classes to see what methods need to be implemented to make a custom container with that behavior:

```
>>> import collections
>>> collections.Sequence()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Sequence with abstract methods \
```

```
__getitem__, __len__
>>>
```

Here is a simple example of a class that implements the preceding methods to create a sequence where items are always stored in sorted order (it's not a particularly efficient implementation, but it illustrates the general idea):

```
import collections
import bisect

class SortedItems(collections.Sequence):
    def __init__(self, initial=None):
        self._items = sorted(initial) if initial is None else []

    # Required sequence methods
    def __getitem__(self, index):
        return self._items[index]

    def __len__(self):
        return len(self._items)

    # Method for adding an item in the right location
    def add(self, item):
        bisect.insort(self._items, item)
```

Here's an example of using this class:

```
>>> items = SortedItems([5, 1, 3])
>>> list(items)
[1, 3, 5]
>>> items[0]
1
>>> items[-1]
5
>>> items.add(2)
>>> list(items)
[1, 2, 3, 5]
>>> items.add(-10)
>>> list(items)
[-10, 1, 2, 3, 5]
>>> items[1:4]
[1, 2, 3]
>>> 3 in items
True
>>> len(items)
5
>>> for n in items:
...     print(n)
...
-10
1
2
3
```

```
5
>>>
```

As you can see, instances of `SortedItems` behave exactly like a normal sequence and support all of the usual operations, including indexing, iteration, `len()`, containment (the `in` operator), and even slicing.

As an aside, the `bisect` module used in this recipe is a convenient way to keep items in a list sorted. The `bisect.insort()` inserts an item into a list so that the list remains in order.

## Discussion

Inheriting from one of the abstract base classes in `collections` ensures that your custom container implements all of the required methods expected of the container. However, this inheritance also facilitates type checking.

For example, your custom container will satisfy various type checks like this:

```
>>> items = SortedItems()
>>> import collections
>>> isinstance(items, collections.Iterable)
True
>>> isinstance(items, collections.Sequence)
True
>>> isinstance(items, collections.Container)
True
>>> isinstance(items, collections.Sized)
True
>>> isinstance(items, collections.Mapping)
False
>>>
```

Many of the abstract base classes in `collections` also provide default implementations of common container methods. To illustrate, suppose you have a class that inherits from `collections.MutableSequence`, like this:

```
class Items(collections.MutableSequence):
    def __init__(self, initial=None):
        self._items = list(initial) if initial is None else []

    # Required sequence methods
    def __getitem__(self, index):
        print('Getting:', index)
        return self._items[index]

    def __setitem__(self, index, value):
        print('Setting:', index, value)
        self._items[index] = value

    def __delitem__(self, index):
```

```

        print('Deleting:', index)
        del self._items[index]

    def insert(self, index, value):
        print('Inserting:', index, value)
        self._items.insert(index, value)

    def __len__(self):
        print('Len')
        return len(self._items)

```

If you create an instance of `Items`, you'll find that it supports almost all of the core list methods (e.g., `append()`, `remove()`, `count()`, etc.). These methods are implemented in such a way that they only use the required ones. Here's an interactive session that illustrates this:

```

>>> a = Items([1, 2, 3])
>>> len(a)
Len
3
>>> a.append(4)
Len
Inserting: 3 4
>>> a.append(2)
Len
Inserting: 4 2
>>> a.count(2)
Getting: 0
Getting: 1
Getting: 2
Getting: 3
Getting: 4
Getting: 5
2
>>> a.remove(3)
Getting: 0
Getting: 1
Getting: 2
Deleting: 2
>>>

```

This recipe only provides a brief glimpse into Python's abstract class functionality. The `numbers` module provides a similar collection of abstract classes related to numeric data types. See [Recipe 8.12](#) for more information about making your own abstract base classes.



## 8.15. Delegating Attribute Access

### Problem

You want an instance to delegate attribute access to an internally held instance possibly as an alternative to inheritance or in order to implement a proxy.

### Solution

Simply stated, delegation is a programming pattern where the responsibility for implementing a particular operation is handed off (i.e., delegated) to a different object. In its simplest form, it often looks something like this:

```
class A:
    def spam(self, x):
        pass

    def foo(self):
        pass

class B:
    def __init__(self):
        self._a = A()

    def spam(self, x):
        # Delegate to the internal self._a instance
        return self._a.spam(x)

    def foo(self):
        # Delegate to the internal self._a instance
        return self._a.foo()

    def bar(self):
        pass
```

If there are only a couple of methods to delegate, writing code such as that just given is easy enough. However, if there are many methods to delegate, an alternative approach is to define the `__getattr__()` method, like this:

```
class A:
    def spam(self, x):
        pass

    def foo(self):
        pass

class B:
    def __init__(self):
        self._a = A()
```

```

def bar(self):
    pass

# Expose all of the methods defined on class A
def __getattr__(self, name):
    return getattr(self._a, name)

```

The `__getattr__()` method is kind of like a catch-all for attribute lookup. It's a method that gets called if code tries to access an attribute that doesn't exist. In the preceding code, it would catch access to undefined methods on B and simply delegate them to A. For example:

```

b = B()
b.bar()    # Calls B.bar() (exists on B)
b.spam(42) # Calls B.__getattr__('spam') and delegates to A.spam

```

Another example of delegation is in the implementation of proxies. For example:

```

# A proxy class that wraps around another object, but
# exposes its public attributes

class Proxy:
    def __init__(self, obj):
        self._obj = obj

    # Delegate attribute lookup to internal obj
    def __getattr__(self, name):
        print('getattr:', name)
        return getattr(self._obj, name)

    # Delegate attribute assignment
    def __setattr__(self, name, value):
        if name.startswith('_'):
            super().__setattr__(name, value)
        else:
            print('setattr:', name, value)
            setattr(self._obj, name, value)

    # Delegate attribute deletion
    def __delattr__(self, name):
        if name.startswith('_'):
            super().__delattr__(name)
        else:
            print('delattr:', name)
            delattr(self._obj, name)

```

To use this proxy class, you simply wrap it around another instance. For example:

```

class Spam:
    def __init__(self, x):
        self.x = x
    def bar(self, y):
        print('Spam.bar:', self.x, y)

```

```

# Create an instance
s = Spam(2)

# Create a proxy around it
p = Proxy(s)

# Access the proxy
print(p.x)      # Outputs 2
p.bar(3)        # Outputs "Spam.bar: 2 3"
p.x = 37        # Changes s.x to 37

```

By customizing the implementation of the attribute access methods, you could customize the proxy to behave in different ways (e.g., logging access, only allowing read-only access, etc.).

## Discussion

Delegation is sometimes used as an alternative to inheritance. For example, instead of writing code like this:

```

class A:
    def spam(self, x):
        print('A.spam', x)

    def foo(self):
        print('A.foo')

class B(A):
    def spam(self, x):
        print('B.spam')
        super().spam(x)

    def bar(self):
        print('B.bar')

```

A solution involving delegation would be written as follows:

```

class A:
    def spam(self, x):
        print('A.spam', x)

    def foo(self):
        print('A.foo')

class B:
    def __init__(self):
        self._a = A()

    def spam(self, x):
        print('B.spam', x)
        self._a.spam(x)

```

```
def bar(self):
    print('B.bar')

def __getattr__(self, name):
    return getattr(self._a, name)
```

This use of delegation is often useful in situations where direct inheritance might not make much sense or where you want to have more control of the relationship between objects (e.g., only exposing certain methods, implementing interfaces, etc.).

When using delegation to implement proxies, there are a few additional details to note. First, the `__getattr__()` method is actually a fallback method that only gets called when an attribute is not found. Thus, when attributes of the proxy instance itself are accessed (e.g., the `_obj` attribute), this method would not be triggered. Second, the `__setattr__()` and `__delattr__()` methods need a bit of extra logic added to separate attributes from the proxy instance itself and attributes on the internal object `_obj`. A common convention is for proxies to only delegate to attributes that don't start with a leading underscore (i.e., proxies only expose the “public” attributes of the held instance).

It is also important to emphasize that the `__getattr__()` method usually does not apply to most special methods that start and end with double underscores. For example, consider this class:

```
class ListLike:
    def __init__(self):
        self._items = []
    def __getattr__(self, name):
        return getattr(self._items, name)
```

If you try to make a `ListLike` object, you'll find that it supports the common list methods, such as `append()` and `insert()`. However, it does not support any of the operators like `len()`, item lookup, and so forth. For example:

```
>>> a = ListLike()
>>> a.append(2)
>>> a.insert(0, 1)
>>> a.sort()
>>> len(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'ListLike' has no len()
>>> a[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'ListLike' object does not support indexing
>>>
```

To support the different operators, you have to manually delegate the associated special methods yourself. For example:

```

class ListLike:
    def __init__(self):
        self._items = []
    def __getattr__(self, name):
        return getattr(self._items, name)

    # Added special methods to support certain list operations
    def __len__(self):
        return len(self._items)
    def __getitem__(self, index):
        return self._items[index]
    def __setitem__(self, index, value):
        self._items[index] = value
    def __delitem__(self, index):
        del self._items[index]

```

See [Recipe 11.8](#) for another example of using delegation in the context of creating proxy classes for remote procedure call.

## 8.16. Defining More Than One Constructor in a Class

### Problem

You’re writing a class, but you want users to be able to create instances in more than the one way provided by `__init__()`.

### Solution

To define a class with more than one constructor, you should use a class method. Here is a simple example:

```

import time

class Date:
    # Primary constructor
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    # Alternate constructor
    @classmethod
    def today(cls):
        t = time.localtime()
        return cls(t.tm_year, t.tm_mon, t.tm_mday)

```

To use the alternate constructor, you simply call it as a function, such as `Date.today()`. Here is an example:

```
a = Date(2012, 12, 21)    # Primary
b = Date.today()         # Alternate
```

## Discussion

One of the primary uses of class methods is to define alternate constructors, as shown in this recipe. A critical feature of a class method is that it receives the class as the first argument (`cls`). You will notice that this class is used within the method to create and return the final instance. It is extremely subtle, but this aspect of class methods makes them work correctly with features such as inheritance. For example:

```
class NewDate(Date):
    pass

c = Date.today()    # Creates an instance of Date (cls=Date)
d = NewDate.today() # Creates an instance of NewDate (cls=NewDate)
```

When defining a class with multiple constructors, you should make the `__init__()` function as simple as possible—doing nothing more than assigning attributes from given values. Alternate constructors can then choose to perform advanced operations if needed.

Instead of defining a separate class method, you might be inclined to implement the `__init__()` method in a way that allows for different calling conventions. For example:

```
class Date:
    def __init__(self, *args):
        if len(args) == 0:
            t = time.localtime()
            args = (t.tm_year, t.tm_mon, t.tm_mday)
        self.year, self.month, self.day = args
```

Although this technique works in certain cases, it often leads to code that is hard to understand and difficult to maintain. For example, this implementation won't show useful help strings (with argument names). In addition, code that creates `Date` instances will be less clear. Compare and contrast the following:

```
a = Date(2012, 12, 21)    # Clear. A specific date.
b = Date()                # ??? What does this do?

# Class method version
c = Date.today()          # Clear. Today's date.
```

As shown, the `Date.today()` invokes the regular `Date.__init__()` method by instantiating a `Date()` with suitable year, month, and day arguments. If necessary, instances can be created without ever invoking the `__init__()` method. This is described in the next recipe.

## 8.17. Creating an Instance Without Invoking *init*

### Problem

You need to create an instance, but want to bypass the execution of the `__init__()` method for some reason.

### Solution

A bare uninitialized instance can be created by directly calling the `__new__()` method of a class. For example, consider this class:

```
class Date:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day
```

Here's how you can create a `Date` instance without invoking `__init__()`:

```
>>> d = Date.__new__(Date)
>>> d
<__main__.Date object at 0x1006716d0>
>>> d.year
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Date' object has no attribute 'year'
>>>
```

As you can see, the resulting instance is uninitialized. Thus, it is now your responsibility to set the appropriate instance variables. For example:

```
>>> data = {'year':2012, 'month':8, 'day':29}
>>> for key, value in data.items():
...     setattr(d, key, value)
...
>>> d.year
2012
>>> d.month
8
>>>
```

### Discussion

The problem of bypassing `__init__()` sometimes arises when instances are being created in a nonstandard way such as when deserializing data or in the implementation of a class method that's been defined as an alternate constructor. For example, on the `Date` class shown, someone might define an alternate constructor `today()` as follows:

```

from time import localtime

class Date:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    @classmethod
    def today(cls):
        d = cls.__new__(cls)
        t = localtime()
        d.year = t.tm_year
        d.month = t.tm_mon
        d.day = t.tm_mday
        return d

```

Similarly, suppose you are deserializing JSON data and, as a result, produce a dictionary like this:

```
data = { 'year': 2012, 'month': 8, 'day': 29 }
```

If you want to turn this into a `Date` instance, simply use the technique shown in the solution.

When creating instances in a nonstandard way, it's usually best to not make too many assumptions about their implementation. As such, you generally don't want to write code that directly manipulates the underlying instance dictionary `__dict__` unless you know it's guaranteed to be defined. Otherwise, the code will break if the class uses `__slots__`, properties, descriptors, or other advanced techniques. By using `setattr()` to set the values, your code will be as general purpose as possible.

## 8.18. Extending Classes with Mixins

### Problem

You have a collection of generally useful methods that you would like to make available for extending the functionality of other class definitions. However, the classes where the methods might be added aren't necessarily related to one another via inheritance. Thus, you can't just attach the methods to a common base class.

### Solution

The problem addressed by this recipe often arises in code where one is interested in the issue of class customization. For example, maybe a library provides a basic set of classes along with a set of optional customizations that can be applied if desired by the user.



To illustrate, suppose you have an interest in adding various customizations (e.g., logging, set-once, type checking, etc.) to mapping objects. Here are a set of mixin classes that do that:

```
class LoggedMappingMixin:
    """
    Add logging to get/set/delete operations for debugging.
    """
    __slots__ = ()

    def __getitem__(self, key):
        print('Getting ' + str(key))
        return super().__getitem__(key)

    def __setitem__(self, key, value):
        print('Setting {} = {}'.format(key, value))
        return super().__setitem__(key, value)

    def __delitem__(self, key):
        print('Deleting ' + str(key))
        return super().__delitem__(key)

class SetOnceMappingMixin:
    """
    Only allow a key to be set once.
    """
    __slots__ = ()

    def __setitem__(self, key, value):
        if key in self:
            raise KeyError(str(key) + ' already set')
        return super().__setitem__(key, value)

class StringKeysMappingMixin:
    """
    Restrict keys to strings only
    """
    __slots__ = ()

    def __setitem__(self, key, value):
        if not isinstance(key, str):
            raise TypeError('keys must be strings')
        return super().__setitem__(key, value)
```

These classes, by themselves, are useless. In fact, if you instantiate any one of them, it does nothing useful at all (other than generate exceptions). Instead, they are supposed to be mixed with other mapping classes through multiple inheritance. For example:

```
>>> class LoggedDict(LoggedMappingMixin, dict):
...     pass
...
>>> d = LoggedDict()
>>> d['x'] = 23
Setting x = 23
```

```

>>> d['x']
Getting x
23
>>> del d['x']
Deleting x

>>> from collections import defaultdict
>>> class SetOnceDefaultDict(SetOnceMappingMixin, defaultdict):
...     pass
...
>>> d = SetOnceDefaultDict(list)
>>> d['x'].append(2)
>>> d['y'].append(3)
>>> d['x'].append(10)
>>> d['x'] = 23
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "mixin.py", line 24, in __setitem__
    raise KeyError(str(key) + ' already set')
KeyError: 'x already set'

>>> from collections import OrderedDict
>>> class StringOrderedDict(StringKeysMappingMixin,
...                         SetOnceMappingMixin,
...                         OrderedDict):
...     pass
...
>>> d = StringOrderedDict()
>>> d['x'] = 23
>>> d[42] = 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "mixin.py", line 45, in __setitem__
    ...
TypeError: keys must be strings
>>> d['x'] = 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "mixin.py", line 46, in __setitem__
    __slots__ = ()
  File "mixin.py", line 24, in __setitem__
    if key in self:
KeyError: 'x already set'
>>>

```

In the example, you will notice that the mixins are combined with other existing classes (e.g., dict, defaultdict, OrderedDict), and even one another. When combined, the classes all work together to provide the desired functionality.

## Discussion

Mixin classes appear in various places in the standard library, mostly as a means for extending the functionality of other classes similar to as shown. They are also one of the main uses of multiple inheritance. For instance, if you are writing network code, you can often use the `ThreadingMixIn` from the `socketserver` module to add thread support to other network-related classes. For example, here is a multithreaded XML-RPC server:

```
from xmlrpc.server import SimpleXMLRPCServer
from socketserver import ThreadingMixIn
class ThreadedXMLRPCServer(ThreadingMixIn, SimpleXMLRPCServer):
    pass
```

It is also common to find mixins defined in large libraries and frameworks—again, typically to enhance the functionality of existing classes with optional features in some way.

There is a rich history surrounding the theory of mixin classes. However, rather than getting into all of the details, there are a few important implementation details to keep in mind.

First, mixin classes are never meant to be instantiated directly. For example, none of the classes in this recipe work by themselves. They have to be mixed with another class that implements the required mapping functionality. Similarly, the `ThreadingMixIn` from the `socketserver` library has to be mixed with an appropriate server class—it can't be used all by itself.

Second, mixin classes typically have no state of their own. This means there is no `__init__()` method and no instance variables. In this recipe, the specification of `__slots__ = ()` is meant to serve as a strong hint that the mixin classes do not have their own instance data.

If you are thinking about defining a mixin class that has an `__init__()` method and instance variables, be aware that there is significant peril associated with the fact that the class doesn't know anything about the other classes it's going to be mixed with. Thus, any instance variables created would have to be named in a way that avoids name clashes. In addition, the `__init__()` method would have to be programmed in a way that properly invokes the `__init__()` method of other classes that are mixed in. In general, this is difficult to implement since you know nothing about the argument signatures of the other classes. At the very least, you would have to implement something very general using `*arg`, `**kwargs`. If the `__init__()` of the mixin class took any arguments of its own, those arguments should be specified by keyword only and named in such a way to avoid name collisions with other arguments. Here is one possible implementation of a mixin defining an `__init__()` and accepting a keyword argument:

```

class RestrictKeysMixin:
    def __init__(self, *args, _restrict_key_type, **kwargs):
        self._restrict_key_type = _restrict_key_type
        super().__init__(*args, **kwargs)

    def __setitem__(self, key, value):
        if not isinstance(key, self._restrict_key_type):
            raise TypeError('Keys must be ' + str(self._restrict_key_type))
        super().__setitem__(key, value)

```

Here is an example that shows how this class might be used:

```

>>> class RDict(RestrictKeysMixin, dict):
...     pass
...
>>> d = RDict(_restrict_key_type=str)
>>> e = RDict([('name', 'Dave'), ('n', 37)], _restrict_key_type=str)
>>> f = RDict(name='Dave', n=37, _restrict_key_type=str)
>>> f
{'n': 37, 'name': 'Dave'}
>>> f[42] = 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "mixin.py", line 83, in __setitem__
    raise TypeError('Keys must be ' + str(self._restrict_key_type))
TypeError: Keys must be <class 'str'>
>>>

```

In this example, you'll notice that initializing an `RDict()` still takes the arguments understood by `dict()`. However, there is an extra keyword argument `restrict_key_type` that is provided to the mixin class.

Finally, use of the `super()` function is an essential and critical part of writing mixin classes. In the solution, the classes redefine certain critical methods, such as `__getitem__()` and `__setitem__()`. However, they also need to call the original implementation of those methods. Using `super()` delegates to the next class on the method resolution order (MRO). This aspect of the recipe, however, is not obvious to novices, because `super()` is being used in classes that have no parent (at first glance, it might look like an error). However, in a class definition such as this:

```

class LoggedDict(LoggedMappingMixin, dict):
    pass

```

the use of `super()` in `LoggedMappingMixin` delegates to the next class over in the multiple inheritance list. That is, a call such as `super().__getitem__()` in `LoggedMappingMixin` actually steps over and invokes `dict.__getitem__()`. Without this behavior, the mixin class wouldn't work at all.

An alternative implementation of mixins involves the use of class decorators. For example, consider this code:

```

def LoggedMapping(cls):
    cls.__getitem__ = cls._getitem__
    cls.__setitem__ = cls._setitem__
    cls.__delitem__ = cls._delitem__

    def _getitem__(self, key):
        print('Getting ' + str(key))
        return cls.__getitem__(self, key)

    def _setitem__(self, key, value):
        print('Setting {} = {}'.format(key, value))
        return cls.__setitem__(self, key, value)

    def _delitem__(self, key):
        print('Deleting ' + str(key))
        return cls.__delitem__(self, key)

    cls.__getitem__ = _getitem__
    cls.__setitem__ = _setitem__
    cls.__delitem__ = _delitem__
    return cls

```

This function is applied as a decorator to a class definition. For example:

```

@LoggedMapping
class LoggedDict(dict):
    pass

```

If you try it, you'll find that you get the same behavior, but multiple inheritance is no longer involved. Instead, the decorator has simply performed a bit of surgery on the class definition to replace certain methods. Further details about class decorators can be found in [Recipe 9.12](#).

See [Recipe 8.13](#) for an advanced recipe involving both mixins and class decorators.

## 8.19. Implementing Stateful Objects or State Machines

### Problem

You want to implement a state machine or an object that operates in a number of different states, but don't want to litter your code with a lot of conditionals.

### Solution

In certain applications, you might have objects that operate differently according to some kind of internal state. For example, consider a simple class representing a connection:

```

class Connection:
    def __init__(self):

```

```

        self.state = 'CLOSED'

    def read(self):
        if self.state != 'OPEN':
            raise RuntimeError('Not open')
        print('reading')

    def write(self, data):
        if self.state != 'OPEN':
            raise RuntimeError('Not open')
        print('writing')

    def open(self):
        if self.state == 'OPEN':
            raise RuntimeError('Already open')
        self.state = 'OPEN'

    def close(self):
        if self.state == 'CLOSED':
            raise RuntimeError('Already closed')
        self.state = 'CLOSED'

```

This implementation presents a couple of difficulties. First, the code is complicated by the introduction of many conditional checks for the state. Second, the performance is degraded because common operations (e.g., `read()` and `write()`) always check the state before proceeding.

A more elegant approach is to encode each operational state as a separate class and arrange for the `Connection` class to delegate to the state class. For example:

```

class Connection:
    def __init__(self):
        self.new_state(ClosedConnectionState)

    def new_state(self, newstate):
        self._state = newstate

    # Delegate to the state class
    def read(self):
        return self._state.read(self)

    def write(self, data):
        return self._state.write(self, data)

    def open(self):
        return self._state.open(self)

    def close(self):
        return self._state.close(self)

# Connection state base class
class ConnectionState:

```

```

    @staticmethod
    def read(conn):
        raise NotImplementedError()

    @staticmethod
    def write(conn, data):
        raise NotImplementedError()

    @staticmethod
    def open(conn):
        raise NotImplementedError()

    @staticmethod
    def close(conn):
        raise NotImplementedError()

# Implementation of different states
class ClosedConnectionState(ConnectionState):
    @staticmethod
    def read(conn):
        raise RuntimeError('Not open')

    @staticmethod
    def write(conn, data):
        raise RuntimeError('Not open')

    @staticmethod
    def open(conn):
        conn.new_state(OpenConnectionState)

    @staticmethod
    def close(conn):
        raise RuntimeError('Already closed')

class OpenConnectionState(ConnectionState):
    @staticmethod
    def read(conn):
        print('reading')

    @staticmethod
    def write(conn, data):
        print('writing')

    @staticmethod
    def open(conn):
        raise RuntimeError('Already open')

    @staticmethod
    def close(conn):
        conn.new_state(ClosedConnectionState)

```

Here is an interactive session that illustrates the use of these classes:

```

>>> c = Connection()
>>> c._state
<class '__main__.ClosedConnectionState'>
>>> c.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example.py", line 10, in read
    return self._state.read(self)
  File "example.py", line 43, in read
    raise RuntimeError('Not open')
RuntimeError: Not open
>>> c.open()
>>> c._state
<class '__main__.OpenConnectionState'>
>>> c.read()
reading
>>> c.write('hello')
writing
>>> c.close()
>>> c._state
<class '__main__.ClosedConnectionState'>
>>>

```

## Discussion

Writing code that features a large set of complicated conditionals and intertwined states is hard to maintain and explain. The solution presented here avoids that by splitting the individual states into their own classes.

It might look a little weird, but each state is implemented by a class with static methods, each of which take an instance of `Connection` as the first argument. This design is based on a decision to not store any instance data in the different state classes themselves. Instead, all instance data should be stored on the `Connection` instance. The grouping of states under a common base class is mostly there to help organize the code and to ensure that the proper methods get implemented. The `NotImplementedError` exception raised in base class methods is just there to make sure that subclasses provide an implementation of the required methods. As an alternative, you might consider the use of an abstract base class, as described in [Recipe 8.12](#).

An alternative implementation technique concerns direct manipulation of the `__class__` attribute of instances. Consider this code:

```

class Connection:
    def __init__(self):
        self.new_state(ClosedConnection)

    def new_state(self, newstate):
        self.__class__ = newstate

    def read(self):

```



```

        raise NotImplementedError()

    def write(self, data):
        raise NotImplementedError()

    def open(self):
        raise NotImplementedError()

    def close(self):
        raise NotImplementedError()

class ClosedConnection(Connection):
    def read(self):
        raise RuntimeError('Not open')

    def write(self, data):
        raise RuntimeError('Not open')

    def open(self):
        self.new_state(OpenConnection)

    def close(self):
        raise RuntimeError('Already closed')

class OpenConnection(Connection):
    def read(self):
        print('reading')

    def write(self, data):
        print('writing')

    def open(self):
        raise RuntimeError('Already open')

    def close(self):
        self.new_state(ClosedConnection)

```

The main feature of this implementation is that it eliminates an extra level of indirection. Instead of having separate `Connection` and `ConnectionState` classes, the two classes are merged together into one. As the state changes, the instance will change its type, as shown here:

```

>>> c = Connection()
>>> c
<__main__.ClosedConnection object at 0x1006718d0>
>>> c.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "state.py", line 15, in read
    raise RuntimeError('Not open')
RuntimeError: Not open
>>> c.open()

```

```

>>> c
<__main__.OpenConnection object at 0x1006718d0>
>>> c.read()
reading
>>> c.close()
>>> c
<__main__.ClosedConnection object at 0x1006718d0>
>>>

```

Object-oriented purists might be offended by the idea of simply changing the instance `__class__` attribute. However, it's technically allowed. Also, it might result in slightly faster code since all of the methods on the connection no longer involve an extra delegation step.)

Finally, either technique is useful in implementing more complicated state machines—especially in code that might otherwise feature large `if-elif-else` blocks. For example:

```

# Original implementation
class State:
    def __init__(self):
        self.state = 'A'
    def action(self, x):
        if state == 'A':
            # Action for A
            ...
            state = 'B'
        elif state == 'B':
            # Action for B
            ...
            state = 'C'
        elif state == 'C':
            # Action for C
            ...
            state = 'A'

# Alternative implementation
class State:
    def __init__(self):
        self.new_state(State_A)

    def new_state(self, state):
        self.__class__ = state

    def action(self, x):
        raise NotImplementedError()

class State_A(State):
    def action(self, x):
        # Action for A
        ...
        self.new_state(State_B)

```

```

class State_B(State):
    def action(self, x):
        # Action for B
        ...
        self.new_state(State_C)

class State_C(State):
    def action(self, x):
        # Action for C
        ...
        self.new_state(State_A)

```

This recipe is loosely based on the state design pattern found in *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1995).

## 8.20. Calling a Method on an Object Given the Name As a String

### Problem

You have the name of a method that you want to call on an object stored in a string and you want to execute the method.

### Solution

For simple cases, you might use `getattr()`, like this:

```

import math

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return 'Point({!r},{!r})'.format(self.x, self.y)

    def distance(self, x, y):
        return math.hypot(self.x - x, self.y - y)

p = Point(2, 3)
d = getattr(p, 'distance')(0, 0)    # Calls p.distance(0, 0)

```

An alternative approach is to use `operator.methodcaller()`. For example:

```

import operator
operator.methodcaller('distance', 0, 0)(p)

```

`operator.methodcaller()` may be useful if you want to look up a method by name and supply the same arguments over and over again. For instance, if you need to sort an entire list of points:

```
points = [  
    Point(1, 2),  
    Point(3, 0),  
    Point(10, -3),  
    Point(-5, -7),  
    Point(-1, 8),  
    Point(3, 2)  
]  
  
# Sort by distance from origin (0, 0)  
points.sort(key=operator.methodcaller('distance', 0, 0))
```

## Discussion

Calling a method is actually two separate steps involving an attribute lookup and a function call. Therefore, to call a method, you simply look up the attribute using `getattr()`, as for any other attribute. To invoke the result as a method, simply treat the result of the lookup as a function.

`operator.methodcaller()` creates a callable object, but also fixes any arguments that are going to be supplied to the method. All that you need to do is provide the appropriate `self` argument. For example:

```
>>> p = Point(3, 4)  
>>> d = operator.methodcaller('distance', 0, 0)  
>>> d(p)  
5.0  
>>>
```

Invoking methods using names contained in strings is somewhat common in code that emulates case statements or variants of the visitor pattern. See the next recipe for a more advanced example.

## 8.21. Implementing the Visitor Pattern

### Problem

You need to write code that processes or navigates through a complicated data structure consisting of many different kinds of objects, each of which needs to be handled in a different way. For example, walking through a tree structure and performing different actions depending on what kind of tree nodes are encountered.

## Solution

The problem addressed by this recipe is one that often arises in programs that build data structures consisting of a large number of different kinds of objects. To illustrate, suppose you are trying to write a program that represents mathematical expressions. To do that, the program might employ a number of classes, like this:

```
class Node:
    pass

class UnaryOperator(Node):
    def __init__(self, operand):
        self.operand = operand

class BinaryOperator(Node):
    def __init__(self, left, right):
        self.left = left
        self.right = right

class Add(BinaryOperator):
    pass

class Sub(BinaryOperator):
    pass

class Mul(BinaryOperator):
    pass

class Div(BinaryOperator):
    pass

class Negate(UnaryOperator):
    pass

class Number(Node):
    def __init__(self, value):
        self.value = value
```

These classes would then be used to build up nested data structures, like this:

```
# Representation of 1 + 2 * (3 - 4) / 5
t1 = Sub(Number(3), Number(4))
t2 = Mul(Number(2), t1)
t3 = Div(t2, Number(5))
t4 = Add(Number(1), t3)
```

The problem is not the creation of such structures, but in writing code that processes them later. For example, given such an expression, a program might want to do any number of things (e.g., produce output, generate instructions, perform translation, etc.).

To enable general-purpose processing, a common solution is to implement the so-called “visitor pattern” using a class similar to this:

```

class NodeVisitor:
    def visit(self, node):
        methname = 'visit_' + type(node).__name__
        meth = getattr(self, methname, None)
        if meth is None:
            meth = self.generic_visit
        return meth(node)

    def generic_visit(self, node):
        raise RuntimeError('No {} method'.format('visit_' + type(node).__name__))

```

To use this class, a programmer inherits from it and implements various methods of the form `visit_Name()`, where `Name` is substituted with the node type. For example, if you want to evaluate the expression, you could write this:

```

class Evaluator(NodeVisitor):
    def visit_Number(self, node):
        return node.value

    def visit_Add(self, node):
        return self.visit(node.left) + self.visit(node.right)

    def visit_Sub(self, node):
        return self.visit(node.left) - self.visit(node.right)

    def visit_Mul(self, node):
        return self.visit(node.left) * self.visit(node.right)

    def visit_Div(self, node):
        return self.visit(node.left) / self.visit(node.right)

    def visit_Negate(self, node):
        return -node.operand

```

Here is an example of how you would use this class using the previously generated expression:

```

>>> e = Evaluator()
>>> e.visit(t4)
0.6
>>>

```

As a completely different example, here is a class that translates an expression into operations on a simple stack machine:

```

class StackCode(NodeVisitor):
    def generate_code(self, node):
        self.instructions = []
        self.visit(node)
        return self.instructions

    def visit_Number(self, node):
        self.instructions.append(('PUSH', node.value))

```

```

def binop(self, node, instruction):
    self.visit(node.left)
    self.visit(node.right)
    self.instructions.append((instruction,))

def visit_Add(self, node):
    self.binop(node, 'ADD')

def visit_Sub(self, node):
    self.binop(node, 'SUB')

def visit_Mul(self, node):
    self.binop(node, 'MUL')

def visit_Div(self, node):
    self.binop(node, 'DIV')

def unaryop(self, node, instruction):
    self.visit(node.operand)
    self.instructions.append((instruction,))

def visit_Negate(self, node):
    self.unaryop(node, 'NEG')

```

Here is an example of this class in action:

```

>>> s = StackCode()
>>> s.generate_code(t4)
[('PUSH', 1), ('PUSH', 2), ('PUSH', 3), ('PUSH', 4), ('SUB',),
 ('MUL',), ('PUSH', 5), ('DIV',), ('ADD',)]
>>>

```

## Discussion

There are really two key ideas in this recipe. The first is a design strategy where code that manipulates a complicated data structure is decoupled from the data structure itself. That is, in this recipe, none of the various `Node` classes provide any implementation that does anything with the data. Instead, all of the data manipulation is carried out by specific implementations of the separate `NodeVisitor` class. This separation makes the code extremely general purpose.

The second major idea of this recipe is in the implementation of the visitor class itself. In the visitor, you want to dispatch to a different handling method based on some value such as the node type. In a naive implementation, you might be inclined to write a huge `if` statement, like this:

```

class NodeVisitor:
    def visit(self, node):
        nodetype = type(node).__name__
        if nodetype == 'Number':

```

```

        return self.visit_Number(node)
    elif nodetype == 'Add':
        return self.visit_Add(node)
    elif nodetype == 'Sub':
        return self.visit_Sub(node)
    ...

```

However, it quickly becomes apparent that you don't really want to take that approach. Aside from being incredibly verbose, it runs slowly, and it's hard to maintain if you ever add or change the kind of nodes being handled. Instead, it's much better to play a little trick where you form the name of a method and go fetch it with the `getattr()` function, as shown. The `generic_visit()` method in the solution is a fallback should no matching handler method be found. In this recipe, it raises an exception to alert the programmer that an unexpected node type was encountered.

Within each visitor class, it is common for calculations to be driven by recursive calls to the `visit()` method. For example:

```

class Evaluator(NodeVisitor):
    ...
    def visit_Add(self, node):
        return self.visit(node.left) + self.visit(node.right)

```

This recursion is what makes the visitor class traverse the entire data structure. Essentially, you keep calling `visit()` until you reach some sort of terminal node, such as `Number` in the example. The exact order of the recursion and other operations depend entirely on the application.

It should be noted that this particular technique of dispatching to a method is also a common way to emulate the behavior of switch or case statements from other languages. For example, if you are writing an HTTP framework, you might have classes that do a similar kind of dispatch:

```

class HTTPHandler:
    def handle(self, request):
        methname = 'do_' + request.request_method
        getattr(self, methname)(request)

    def do_GET(self, request):
        ...
    def do_POST(self, request):
        ...
    def do_HEAD(self, request):
        ...

```

One weakness of the visitor pattern is its heavy reliance on recursion. If you try to apply it to a deeply nested structure, it's possible that you will hit Python's recursion depth limit (see `sys.getrecursionlimit()`). To avoid this problem, you can make certain choices in your data structures. For example, you can use normal Python lists instead of linked lists or try to aggregate more data in each node to make the data more shallow.



You can also try to employ nonrecursive traversal algorithms using generators or iterators as discussed in [Recipe 8.22](#).

Use of the visitor pattern is extremely common in programs related to parsing and compiling. One notable implementation can be found in Python's own `ast` module. In addition to allowing traversal of tree structures, it provides a variation that allows a data structure to be rewritten or transformed as it is traversed (e.g., nodes added or removed). Look at the source for `ast` for more details. [Recipe 9.24](#) shows an example of using the `ast` module to process Python source code.

## 8.22. Implementing the Visitor Pattern Without Recursion

### Problem

You're writing code that navigates through a deeply nested tree structure using the visitor pattern, but it blows up due to exceeding the recursion limit. You'd like to eliminate the recursion, but keep the programming style of the visitor pattern.

### Solution

Clever use of generators can sometimes be used to eliminate recursion from algorithms involving tree traversal or searching. In [Recipe 8.21](#), a visitor class was presented. Here is an alternative implementation of that class that drives the computation in an entirely different way using a stack and generators:

```
import types

class Node:
    pass

import types
class NodeVisitor:
    def visit(self, node):
        stack = [ node ]
        last_result = None
        while stack:
            try:
                last = stack[-1]
                if isinstance(last, types.GeneratorType):
                    stack.append(last.send(last_result))
                    last_result = None
                elif isinstance(last, Node):
                    stack.append(self._visit(stack.pop()))
                else:
                    last_result = stack.pop()
            except StopIteration:
                stack.pop()
        return last_result
```

```

def _visit(self, node):
    methname = 'visit_' + type(node).__name__
    meth = getattr(self, methname, None)
    if meth is None:
        meth = self.generic_visit
    return meth(node)

def generic_visit(self, node):
    raise RuntimeError('No {} method'.format('visit_' + type(node).__name__))

```

If you use this class, you'll find that it still works with existing code that might have used recursion. In fact, you can use it as a drop-in replacement for the visitor implementation in the prior recipe. For example, consider the following code, which involves expression trees:

```

class UnaryOperator(Node):
    def __init__(self, operand):
        self.operand = operand

class BinaryOperator(Node):
    def __init__(self, left, right):
        self.left = left
        self.right = right

class Add(BinaryOperator):
    pass

class Sub(BinaryOperator):
    pass

class Mul(BinaryOperator):
    pass

class Div(BinaryOperator):
    pass

class Negate(UnaryOperator):
    pass

class Number(Node):
    def __init__(self, value):
        self.value = value

# A sample visitor class that evaluates expressions
class Evaluator(NodeVisitor):
    def visit_Number(self, node):
        return node.value

    def visit_Add(self, node):
        return self.visit(node.left) + self.visit(node.right)

```

```

def visit_Sub(self, node):
    return self.visit(node.left) - self.visit(node.right)

def visit_Mul(self, node):
    return self.visit(node.left) * self.visit(node.right)

def visit_Div(self, node):
    return self.visit(node.left) / self.visit(node.right)

def visit_Negate(self, node):
    return -self.visit(node.operand)

if __name__ == '__main__':
    # 1 + 2*(3-4) / 5
    t1 = Sub(Number(3), Number(4))
    t2 = Mul(Number(2), t1)
    t3 = Div(t2, Number(5))
    t4 = Add(Number(1), t3)

    # Evaluate it
    e = Evaluator()
    print(e.visit(t4))    # Outputs 0.6

```

The preceding code works for simple expressions. However, the implementation of Evaluator uses recursion and crashes if things get too nested. For example:

```

>>> a = Number(0)
>>> for n in range(1, 100000):
...     a = Add(a, Number(n))
...
>>> e = Evaluator()
>>> e.visit(a)
Traceback (most recent call last):
...
File "visitor.py", line 29, in _visit
    return meth(node)
File "visitor.py", line 67, in visit_Add
    return self.visit(node.left) + self.visit(node.right)
RuntimeError: maximum recursion depth exceeded
>>>

```

Now let's change the Evaluator class ever so slightly to the following:

```

class Evaluator(NodeVisitor):
    def visit_Number(self, node):
        return node.value

    def visit_Add(self, node):
        yield (yield node.left) + (yield node.right)

    def visit_Sub(self, node):
        yield (yield node.left) - (yield node.right)

```

```

def visit_Mul(self, node):
    yield (yield node.left) * (yield node.right)

def visit_Div(self, node):
    yield (yield node.left) / (yield node.right)

def visit_Negate(self, node):
    yield -(yield node.operand)

```

If you try the same recursive experiment, you'll find that it suddenly works. It's magic!

```

>>> a = Number(0)
>>> for n in range(1,100000):
...     a = Add(a, Number(n))
...
>>> e = Evaluator()
>>> e.visit(a)
4999950000
>>>

```

If you want to add custom processing into any of the methods, it still works. For example:

```

class Evaluator(NodeVisitor):
    ...
    def visit_Add(self, node):
        print('Add:', node)
        lhs = yield node.left
        print('left=', lhs)
        rhs = yield node.right
        print('right=', rhs)
        yield lhs + rhs
    ...

```

Here is some sample output:

```

>>> e = Evaluator()
>>> e.visit(t4)
Add: <__main__.Add object at 0x1006a8d90>
left= 1
right= -0.4
0.6
>>>

```

## Discussion

This recipe nicely illustrates how generators and coroutines can perform mind-bending tricks involving program control flow, often to great advantage. To understand this recipe, a few key insights are required.

First, in problems related to tree traversal, a common implementation strategy for avoiding recursion is to write algorithms involving a stack or queue. For example, depth-first traversal can be implemented entirely by pushing nodes onto a stack when first encountered and then popping them off once processing has finished. The central core

of the `visit()` method in the solution is built around this idea. The algorithm starts by pushing the initial node onto the `stack` list and runs until the stack is empty. During execution, the stack will grow according to the depth of the underlying tree structure.

The second insight concerns the behavior of the `yield` statement in generators. When `yield` is encountered, the behavior of a generator is to emit a value and to suspend. This recipe uses this as a replacement for recursion. For example, instead of writing a recursive expression like this:

```
value = self.visit(node.left)
```

you replace it with the following:

```
value = yield node.left
```

Behind the scenes, this sends the node in question (`node.left`) back to the `visit()` method. The `visit()` method then carries out the execution of the appropriate `visit_Name()` method for that node. In some sense, this is almost the opposite of recursion. That is, instead of calling `visit()` recursively to move the algorithm forward, the `yield` statement is being used to temporarily back out of the computation in progress. Thus, the `yield` is essentially a signal that tells the algorithm that the yielded node needs to be processed first before further progress can be made.

The final part of this recipe concerns propagation of results. When generator functions are used, you can no longer use `return` statements to emit values (doing so will cause a `SyntaxError` exception). Thus, the `yield` statement has to do double duty to cover the case. In this recipe, if the value produced by a `yield` statement is a non-Node type, it is assumed to be a value that will be propagated to the next step of the calculation. This is the purpose of the `last_return` variable in the code. Typically, this would hold the last value yielded by a visit method. That value would then be sent into the previously executing method, where it would show up as the return value from a `yield` statement. For example, in this code:

```
value = yield node.left
```

The `value` variable gets the value of `last_return`, which is the result returned by the visitor method invoked for `node.left`.

All of these aspects of the recipe are found in this fragment of code:

```
try:
    last = stack[-1]
    if isinstance(last, types.GeneratorType):
        stack.append(last.send(last_result))
        last_result = None
    elif isinstance(last, Node):
        stack.append(self._visit(stack.pop()))
    else:
        last_result = stack.pop()
```

```
except StopIteration:
    stack.pop()
```

The code works by simply looking at the top of the stack and deciding what to do next. If it's a generator, then its `send()` method is invoked with the last result (if any) and the result appended onto the stack for further processing. The value returned by `send()` is the same value that was given to the `yield` statement. Thus, in a statement such as `yield node.left`, the `Node` instance `node.left` is returned by `send()` and placed on the top of the stack.

If the top of the stack is a `Node` instance, then it is replaced by the result of calling the appropriate visit method for that node. This is where the underlying recursion is being eliminated. Instead of the various visit methods directly calling `visit()` recursively, it takes place here. As long as the methods use `yield`, it all works out.

Finally, if the top of the stack is anything else, it's assumed to be a return value of some kind. It just gets popped off the stack and placed into `last_result`. If the next item on the stack is a generator, then it gets sent in as a return value for the `yield`. It should be noted that the final return value of `visit()` is also set to `last_result`. This is what makes this recipe work with a traditional recursive implementation. If no generators are being used, this value simply holds the value given to any `return` statements used in the code.

One potential danger of this recipe concerns the distinction between yielding `Node` and non-`Node` values. In the implementation, all `Node` instances are automatically traversed. This means that you can't use a `Node` as a return value to be propagated. In practice, this may not matter. However, if it does, you might need to adapt the algorithm slightly. For example, possibly by introducing another class into the mix, like this:

```
class Visit:
    def __init__(self, node):
        self.node = node

class NodeVisitor:
    def visit(self, node):
        stack = [ Visit(node) ]
        last_result = None
        while stack:
            try:
                last = stack[-1]
                if isinstance(last, types.GeneratorType):
                    stack.append(last.send(last_result))
                    last_result = None
                elif isinstance(last, Visit):
                    stack.append(self._visit(stack.pop().node))
                else:
                    last_result = stack.pop()
            except StopIteration:
```

```

        stack.pop()
    return last_result

def _visit(self, node):
    methname = 'visit_' + type(node).__name__
    meth = getattr(self, methname, None)
    if meth is None:
        meth = self.generic_visit
    return meth(node)

def generic_visit(self, node):
    raise RuntimeError('No {} method'.format('visit_' + type(node).__name__))

```

With this implementation, the various visitor methods would now look like this:

```

class Evaluator(NodeVisitor):
    ...
    def visit_Add(self, node):
        yield (yield Visit(node.left)) + (yield Visit(node.right))

    def visit_Sub(self, node):
        yield (yield Visit(node.left)) - (yield Visit(node.right))
    ...

```

Having seen this recipe, you might be inclined to investigate a solution that doesn't involve `yield`. However, doing so will lead to code that has to deal with many of the same issues presented here. For example, to eliminate recursion, you'll need to maintain a stack. You'll also need to come up with some scheme for managing the traversal and invoking various visitor-related logic. Without generators, this code ends up being a very messy mix of stack manipulation, callback functions, and other constructs. Frankly, the main benefit of using `yield` is that you can write nonrecursive code in an elegant style that looks almost exactly like the recursive implementation.

## 8.23. Managing Memory in Cyclic Data Structures

### Problem

Your program creates data structures with cycles (e.g., trees, graphs, observer patterns, etc.), but you are experiencing problems with memory management.

### Solution

A simple example of a cyclic data structure is a tree structure where a parent points to its children and the children point back to their parent. For code like this, you should consider making one of the links a weak reference using the `weakref` library. For example:

```

import weakref

class Node:
    def __init__(self, value):
        self.value = value
        self._parent = None
        self.children = []

    def __repr__(self):
        return 'Node({!r:})'.format(self.value)

    # property that manages the parent as a weak-reference
    @property
    def parent(self):
        return self._parent if self._parent is None else self._parent()

    @parent.setter
    def parent(self, node):
        self._parent = weakref.ref(node)

    def add_child(self, child):
        self.children.append(child)
        child.parent = self

```

This implementation allows the parent to quietly die. For example:

```

>>> root = Node('parent')
>>> c1 = Node('child')
>>> root.add_child(c1)
>>> print(c1.parent)
Node('parent')
>>> del root
>>> print(c1.parent)
None
>>>

```

## Discussion

Cyclic data structures are a somewhat tricky aspect of Python that require careful study because the usual rules of garbage collection often don't apply. For example, consider this code:

```

# Class just to illustrate when deletion occurs
class Data:
    def __del__(self):
        print('Data.__del__')

# Node class involving a cycle
class Node:
    def __init__(self):
        self.data = Data()
        self.parent = None

```



```

        self.children = []
    def add_child(self, child):
        self.children.append(child)
        child.parent = self

```

Now, using this code, try some experiments to see some subtle issues with garbage collection:

```

>>> a = Data()
>>> del a           # Immediately deleted
Data.__del__
>>> a = Node()
>>> del a           # Immediately deleted
Data.__del__
>>> a = Node()
>>> a.add_child(Node())
>>> del a           # Not deleted (no message)
>>>

```

As you can see, objects are deleted immediately all except for the last case involving a cycle. The reason is that Python's garbage collection is based on simple reference counting. When the reference count of an object reaches 0, it is immediately deleted. For cyclic data structures, however, this never happens. Thus, in the last part of the example, the parent and child nodes refer to each other, keeping the reference count nonzero.

To deal with cycles, there is a separate garbage collector that runs periodically. However, as a general rule, you never know when it might run. Consequently, you never really know when cyclic data structures might get collected. If necessary, you can force garbage collection, but doing so is a bit clunky:

```

>>> import gc
>>> gc.collect()    # Force collection
Data.__del__
Data.__del__
>>>

```

An even worse problem occurs if the objects involved in a cycle define their own `__del__()` method. For example, suppose the code looked like this:

```

# Class just to illustrate when deletion occurs
class Data:
    def __del__(self):
        print('Data.__del__')

# Node class involving a cycle
class Node:
    def __init__(self):
        self.data = Data()
        self.parent = None
        self.children = []

# NEVER DEFINE LIKE THIS.

```

```

# Only here to illustrate pathological behavior
def __del__(self):
    del self.data
    del self.parent
    del self.children

def add_child(self, child):
    self.children.append(child)
    child.parent = self

```

In this case, the data structures will never be garbage collected at all and your program will leak memory! If you try it, you'll see that the `Data.__del__` message never appears at all—even after a forced garbage collection:

```

>>> a = Node()
>>> a.add_child(Node())
>>> del a           # No message (not collected)
>>> import gc
>>> gc.collect()    # No message (not collected)
>>>

```

Weak references solve this problem by eliminating reference cycles. Essentially, a weak reference is a pointer to an object that does not increase its reference count. You create weak references using the `weakref` library. For example:

```

>>> import weakref
>>> a = Node()
>>> a_ref = weakref.ref(a)
>>> a_ref
<weakref at 0x100581f70; to 'Node' at 0x1005c5410>
>>>

```

To dereference a weak reference, you call it like a function. If the referenced object still exists, it is returned. Otherwise, `None` is returned. Since the reference count of the original object wasn't increased, it can be deleted normally. For example:

```

>>> print(a_ref())
<__main__.Node object at 0x1005c5410>
>>> del a
Data.__del__
>>> print(a_ref())
None
>>>

```

By using weak references, as shown in the solution, you'll find that there are no longer any reference cycles and that garbage collection occurs immediately once a node is no longer being used. See [Recipe 8.25](#) for another example involving weak references.

## 8.24. Making Classes Support Comparison Operations

### Problem

You'd like to be able to compare instances of your class using the standard comparison operators (e.g., `>=`, `!=`, `<=`, etc.), but without having to write a lot of special methods.

### Solution

Python classes can support comparison by implementing a special method for each comparison operator. For example, to support the `>=` operator, you define a `__ge__()` method in the classes. Although defining a single method is usually no problem, it quickly gets tedious to create implementations of every possible comparison operator.

The `functools.total_ordering` decorator can be used to simplify this process. To use it, you decorate a class with it, and define `__eq__()` and one other comparison method (`__lt__`, `__le__`, `__gt__`, or `__ge__`). The decorator then fills in the other comparison methods for you.

As an example, let's build some houses and add some rooms to them, and then perform comparisons based on the size of the houses:

```
from functools import total_ordering

class Room:
    def __init__(self, name, length, width):
        self.name = name
        self.length = length
        self.width = width
        self.square_feet = self.length * self.width

@total_ordering
class House:
    def __init__(self, name, style):
        self.name = name
        self.style = style
        self.rooms = list()

    @property
    def living_space_footage(self):
        return sum(r.square_feet for r in self.rooms)

    def add_room(self, room):
        self.rooms.append(room)

    def __str__(self):
        return '{}: {} square foot {}'.format(self.name,
                                                self.living_space_footage,
                                                self.style)
```

```

def __eq__(self, other):
    return self.living_space_footage == other.living_space_footage

def __lt__(self, other):
    return self.living_space_footage < other.living_space_footage

```

Here, the House class has been decorated with `@total_ordering`. Definitions of `__eq__()` and `__lt__()` are provided to compare houses based on the total square footage of their rooms. This minimum definition is all that is required to make all of the other comparison operations work. For example:

```

# Build a few houses, and add rooms to them
h1 = House('h1', 'Cape')
h1.add_room(Room('Master Bedroom', 14, 21))
h1.add_room(Room('Living Room', 18, 20))
h1.add_room(Room('Kitchen', 12, 16))
h1.add_room(Room('Office', 12, 12))

h2 = House('h2', 'Ranch')
h2.add_room(Room('Master Bedroom', 14, 21))
h2.add_room(Room('Living Room', 18, 20))
h2.add_room(Room('Kitchen', 12, 16))

h3 = House('h3', 'Split')
h3.add_room(Room('Master Bedroom', 14, 21))
h3.add_room(Room('Living Room', 18, 20))
h3.add_room(Room('Office', 12, 16))
h3.add_room(Room('Kitchen', 15, 17))
houses = [h1, h2, h3]

print('Is h1 bigger than h2?', h1 > h2) # prints True
print('Is h2 smaller than h3?', h2 < h3) # prints True
print('Is h2 greater than or equal to h1?', h2 >= h1) # Prints False
print('Which one is biggest?', max(houses)) # Prints 'h3: 1101-square-foot Split'
print('Which is smallest?', min(houses)) # Prints 'h2: 846-square-foot Ranch'

```

## Discussion

If you've written the code to make a class support all of the basic comparison operators, then `total_ordering` probably doesn't seem all that magical: it literally defines a mapping from each of the comparison-supporting methods to all of the other ones that would be required. So, if you defined `__lt__()` in your class as in the solution, it is used to build all of the other comparison operators. It's really just filling in the class with methods like this:

```

class House:
    def __eq__(self, other):
        ...
    def __lt__(self, other):
        ...

```

```
# Methods created by @total_ordering
__le__ = lambda self, other: self < other or self == other
__gt__ = lambda self, other: not (self < other or self == other)
__ge__ = lambda self, other: not (self < other)
__ne__ = lambda self, other: not self == other
```

Sure, it's not hard to write these methods yourself, but `@total_ordering` simply takes the guesswork out of it.

## 8.25. Creating Cached Instances

### Problem

When creating instances of a class, you want to return a cached reference to a previous instance created with the same arguments (if any).

### Solution

The problem being addressed in this recipe sometimes arises when you want to ensure that there is only one instance of a class created for a set of input arguments. Practical examples include the behavior of libraries, such as the `logging` module, that only want to associate a single logger instance with a given name. For example:

```
>>> import logging
>>> a = logging.getLogger('foo')
>>> b = logging.getLogger('bar')
>>> a is b
False
>>> c = logging.getLogger('foo')
>>> a is c
True
>>>
```

To implement this behavior, you should make use of a factory function that's separate from the class itself. For example:

```
# The class in question
class Spam:
    def __init__(self, name):
        self.name = name

# Caching support
import weakref
_spam_cache = weakref.WeakValueDictionary()

def get_spam(name):
    if name not in _spam_cache:
        s = Spam(name)
        _spam_cache[name] = s
    else:
```

```

        s = _spam_cache[name]
    return s

```

If you use this implementation, you'll find that it behaves in the manner shown earlier:

```

>>> a = get_spam('foo')
>>> b = get_spam('bar')
>>> a is b
False
>>> c = get_spam('foo')
>>> a is c
True
>>>

```

## Discussion

Writing a special factory function is often a simple approach for altering the normal rules of instance creation. One question that often arises at this point is whether or not a more elegant approach could be taken.

For example, you might consider a solution that redefines the `__new__()` method of a class as follows:

```

# Note: This code doesn't quite work
import weakref

class Spam:
    _spam_cache = weakref.WeakValueDictionary()
    def __new__(cls, name):
        if name in cls._spam_cache:
            return cls._spam_cache[name]
        else:
            self = super().__new__(cls)
            cls._spam_cache[name] = self
            return self

    def __init__(self, name):
        print('Initializing Spam')
        self.name = name

```

At first glance, it seems like this code might do the job. However, a major problem is that the `__init__()` method always gets called, regardless of whether the instance was cached or not. For example:

```

>>> s = Spam('Dave')
Initializing Spam
>>> t = Spam('Dave')
Initializing Spam
>>> s is t
True
>>>

```

That behavior is probably not what you want. So, to solve the problem of caching without reinitialization, you need to take a slightly different approach.

The use of weak references in this recipe serves an important purpose related to garbage collection, as described in [Recipe 8.23](#). When maintaining a cache of instances, you often only want to keep items in the cache as long as they're actually being used somewhere in the program. A `WeakValueDictionary` instance only holds onto the referenced items as long as they exist somewhere else. Otherwise, the dictionary keys disappear when instances are no longer being used. Observe:

```
>>> a = get_spam('foo')
>>> b = get_spam('bar')
>>> c = get_spam('foo')
>>> list(_spam_cache)
['foo', 'bar']
>>> del a
>>> del c
>>> list(_spam_cache)
['bar']
>>> del b
>>> list(_spam_cache)
[]
>>>
```

For many programs, the bare-bones code shown in this recipe will often suffice. However, there are a number of more advanced implementation techniques that can be considered.

One immediate concern with this recipe might be its reliance on global variables and a factory function that's decoupled from the original class definition. One way to clean this up is to put the caching code into a separate manager class and glue things together like this:

```
import weakref

class CachedSpamManager:
    def __init__(self):
        self._cache = weakref.WeakValueDictionary()
    def get_spam(self, name):
        if name not in self._cache:
            s = Spam(name)
            self._cache[name] = s
        else:
            s = self._cache[name]
        return s
    def clear(self):
        self._cache.clear()

class Spam:
    manager = CachedSpamManager()
```

```

def __init__(self, name):
    self.name = name

def get_spam(name):
    return Spam.manager.get_spam(name)

```

One feature of this approach is that it affords a greater degree of potential flexibility. For example, different kinds of management schemes could be implemented (as separate classes) and attached to the `Spam` class as a replacement for the default caching implementation. None of the other code (e.g., `get_spam`) would need to be changed to make it work.

Another design consideration is whether or not you want to leave the class definition exposed to the user. If you do nothing, a user can easily make instances, bypassing the caching mechanism:

```

>>> a = Spam('foo')
>>> b = Spam('foo')
>>> a is b
False
>>>

```

If preventing this is important, you can take certain steps to avoid it. For example, you might give the class a name starting with an underscore, such as `_Spam`, which at least gives the user a clue that they shouldn't access it directly.

Alternatively, if you want to give users a stronger hint that they shouldn't instantiate `Spam` instances directly, you can make `__init__()` raise an exception and use a class method to make an alternate constructor like this:

```

class Spam:
    def __init__(self, *args, **kwargs):
        raise RuntimeError("Can't instantiate directly")

    # Alternate constructor
    @classmethod
    def _new(cls, name):
        self = cls.__new__(cls)
        self.name = name

```

To use this, you modify the caching code to use `Spam._new()` to create instances instead of the usual call to `Spam()`. For example:

```

import weakref

class CachedSpamManager:
    def __init__(self):
        self._cache = weakref.WeakValueDictionary()
    def get_spam(self, name):
        if name not in self._cache:
            s = Spam._new(name)           # Modified creation

```



```
        self._cache[name] = s
    else:
        s = self._cache[name]
    return s
```

Although there are more extreme measures that can be taken to hide the visibility of the `Spam` class, it's probably best to not overthink the problem. Using an underscore on the name or defining a class method constructor is usually enough for programmers to get a hint.

Caching and other creational patterns can often be solved in a more elegant (albeit advanced) manner through the use of metaclasses. See [Recipe 9.13](#).



---

# Metaprogramming

One of the most important mantras of software development is “don’t repeat yourself.” That is, any time you are faced with a problem of creating highly repetitive code (or cutting or pasting source code), it often pays to look for a more elegant solution. In Python, such problems are often solved under the category of “metaprogramming.” In a nutshell, metaprogramming is about creating functions and classes whose main goal is to manipulate code (e.g., modifying, generating, or wrapping existing code). The main features for this include decorators, class decorators, and metaclasses. However, a variety of other useful topics—including signature objects, execution of code with `exec()`, and inspecting the internals of functions and classes—enter the picture. The main purpose of this chapter is to explore various metaprogramming techniques and to give examples of how they can be used to customize the behavior of Python to your own whims.

## 9.1. Putting a Wrapper Around a Function

### Problem

You want to put a wrapper layer around a function that adds extra processing (e.g., logging, timing, etc.).

### Solution

If you ever need to wrap a function with extra code, define a decorator function. For example:

```
import time
from functools import wraps

def timethis(func):
    '''
    Decorator that reports the execution time.
```

```

'''
@wraps(func)
def wrapper(*args, **kwargs):
    start = time.time()
    result = func(*args, **kwargs)
    end = time.time()
    print(func.__name__, end-start)
    return result
return wrapper

```

Here is an example of using the decorator:

```

>>> @timethis
... def countdown(n):
...     '''
...     Counts down
...     '''
...     while n > 0:
...         n -= 1
...
>>> countdown(100000)
countdown 0.008917808532714844
>>> countdown(10000000)
countdown 0.87188299392912
>>>

```

## Discussion

A decorator is a function that accepts a function as input and returns a new function as output. Whenever you write code like this:

```

@timethis
def countdown(n):
    ...

```

it's the same as if you had performed these separate steps:

```

def countdown(n):
    ...

countdown = timethis(countdown)

```

As an aside, built-in decorators such as `@staticmethod`, `@classmethod`, and `@property` work in the same way. For example, these two code fragments are equivalent:

```

class A:
    @classmethod
    def method(cls):
        pass

class B:
    # Equivalent definition of a class method
    def method(cls):

```

```
pass
method = classmethod(method)
```

The code inside a decorator typically involves creating a new function that accepts any arguments using `*args` and `**kwargs`, as shown with the `wrapper()` function in this recipe. Inside this function, you place a call to the original input function and return its result. However, you also place whatever extra code you want to add (e.g., timing). The newly created function `wrapper` is returned as a result and takes the place of the original function.

It's critical to emphasize that decorators generally do not alter the calling signature or return value of the function being wrapped. The use of `*args` and `**kwargs` is there to make sure that any input arguments can be accepted. The return value of a decorator is almost always the result of calling `func(*args, **kwargs)`, where `func` is the original unwrapped function.

When first learning about decorators, it is usually very easy to get started with some simple examples, such as the one shown. However, if you are going to write decorators for real, there are some subtle details to consider. For example, the use of the decorator `@wraps(func)` in the solution is an easy to forget but important technicality related to preserving function metadata, which is described in the next recipe. The next few recipes that follow fill in some details that will be important if you wish to write decorator functions of your own.

## 9.2. Preserving Function Metadata When Writing Decorators

### Problem

You've written a decorator, but when you apply it to a function, important metadata such as the name, doc string, annotations, and calling signature are lost.

### Solution

Whenever you define a decorator, you should always remember to apply the `@wraps` decorator from the `functools` library to the underlying wrapper function. For example:

```
import time
from functools import wraps

def timethis(func):
    """
    Decorator that reports the execution time.
    """
    @wraps(func)
    def wrapper(*args, **kwargs):
```

```

        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(func.__name__, end-start)
        return result
    return wrapper

```

Here is an example of using the decorator and examining the resulting function meta-data:

```

>>> @timethis
... def countdown(n:int):
...     '''
...     Counts down
...     '''
...     while n > 0:
...         n -= 1
...
>>> countdown(100000)
countdown 0.008917808532714844
>>> countdown.__name__
'countdown'
>>> countdown.__doc__
'\n\tCounts down\n\t'
>>> countdown.__annotations__
{'n': <class 'int'>}
>>>

```

## Discussion

Copying decorator metadata is an important part of writing decorators. If you forget to use `@wraps`, you'll find that the decorated function loses all sorts of useful information. For instance, if omitted, the metadata in the last example would look like this:

```

>>> countdown.__name__
'wrapper'
>>> countdown.__doc__
>>> countdown.__annotations__
{}
>>>

```

An important feature of the `@wraps` decorator is that it makes the wrapped function available to you in the `__wrapped__` attribute. For example, if you want to access the wrapped function directly, you could do this:

```

>>> countdown.__wrapped__(100000)
>>>

```

The presence of the `__wrapped__` attribute also makes decorated functions properly expose the underlying signature of the wrapped function. For example:

```
>>> from inspect import signature
>>> print(signature(countdown))
(n:int)
>>>
```

One common question that sometimes arises is how to make a decorator that directly copies the calling signature of the original function being wrapped (as opposed to using `*args` and `**kwargs`). In general, this is difficult to implement without resorting to some trick involving the generator of code strings and `exec()`. Frankly, you're usually best off using `@wraps` and relying on the fact that the underlying function signature can be propagated by access to the underlying `__wrapped__` attribute. See [Recipe 9.16](#) for more information about signatures.

## 9.3. Unwrapping a Decorator

### Problem

A decorator has been applied to a function, but you want to “undo” it, gaining access to the original unwrapped function.

### Solution

Assuming that the decorator has been implemented properly using `@wraps` (see [Recipe 9.2](#)), you can usually gain access to the original function by accessing the `__wrapped__` attribute. For example:

```
>>> @somedecorator
>>> def add(x, y):
...     return x + y
...
>>> orig_add = add.__wrapped__
>>> orig_add(3, 4)
7
>>>
```

### Discussion

Gaining direct access to the unwrapped function behind a decorator can be useful for debugging, introspection, and other operations involving functions. However, this recipe only works if the implementation of a decorator properly copies metadata using `@wraps` from the `functools` module or sets the `__wrapped__` attribute directly.

If multiple decorators have been applied to a function, the behavior of accessing `__wrapped__` is currently undefined and should probably be avoided. In Python 3.3, it bypasses all of the layers. For example, suppose you have code like this:

```

from functools import wraps

def decorator1(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print('Decorator 1')
        return func(*args, **kwargs)
    return wrapper

def decorator2(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print('Decorator 2')
        return func(*args, **kwargs)
    return wrapper

@decorator1
@decorator2
def add(x, y):
    return x + y

```

Here is what happens when you call the decorated function and the original function through `__wrapped__`:

```

>>> add(2, 3)
Decorator 1
Decorator 2
5
>>> add.__wrapped__(2, 3)
5
>>>

```

However, this behavior has been reported as a bug (see <http://bugs.python.org/issue17482>) and may be changed to expose the proper decorator chain in a future release.

Last, but not least, be aware that not all decorators utilize `@wraps`, and thus, they may not work as described. In particular, the built-in decorators `@staticmethod` and `@classmethod` create descriptor objects that don't follow this convention (instead, they store the original function in a `__func__` attribute). Your mileage may vary.

## 9.4. Defining a Decorator That Takes Arguments

### Problem

You want to write a decorator function that takes arguments.



## Solution

Let's illustrate the process of accepting arguments with an example. Suppose you want to write a decorator that adds logging to a function, but allows the user to specify the logging level and other details as arguments. Here is how you might define the decorator:

```
from functools import wraps
import logging

def logged(level, name=None, message=None):
    """
    Add logging to a function. level is the logging
    level, name is the logger name, and message is the
    log message. If name and message aren't specified,
    they default to the function's module and name.
    """
    def decorate(func):
        logname = name if name else func.__module__
        log = logging.getLogger(logname)
        logmsg = message if message else func.__name__

        @wraps(func)
        def wrapper(*args, **kwargs):
            log.log(level, logmsg)
            return func(*args, **kwargs)
        return wrapper
    return decorate

# Example use
@logged(logging.DEBUG)
def add(x, y):
    return x + y

@logged(logging.CRITICAL, 'example')
def spam():
    print('Spam!')
```

On first glance, the implementation looks tricky, but the idea is relatively simple. The outermost function `logged()` accepts the desired arguments and simply makes them available to the inner functions of the decorator. The inner function `decorate()` accepts a function and puts a wrapper around it as normal. The key part is that the wrapper is allowed to use the arguments passed to `logged()`.

## Discussion

Writing a decorator that takes arguments is tricky because of the underlying calling sequence involved. Specifically, if you have code like this:

```
@decorator(x, y, z)
def func(a, b):
    pass
```

The decoration process evaluates as follows:

```
def func(a, b):
    pass

func = decorator(x, y, z)(func)
```

Carefully observe that the result of `decorator(x, y, z)` must be a callable which, in turn, takes a function as input and wraps it. See [Recipe 9.7](#) for another example of a decorator taking arguments.

## 9.5. Defining a Decorator with User Adjustable Attributes

### Problem

You want to write a decorator function that wraps a function, but has user adjustable attributes that can be used to control the behavior of the decorator at runtime.

### Solution

Here is a solution that expands on the last recipe by introducing accessor functions that change internal variables through the use of `nonlocal` variable declarations. The accessor functions are then attached to the wrapper function as function attributes.

```
from functools import wraps, partial
import logging

# Utility decorator to attach a function as an attribute of obj
def attach_wrapper(obj, func=None):
    if func is None:
        return partial(attach_wrapper, obj)
    setattr(obj, func.__name__, func)
    return func

def logged(level, name=None, message=None):
    """
    Add logging to a function. level is the logging
    level, name is the logger name, and message is the
    log message. If name and message aren't specified,
    they default to the function's module and name.
    """
    def decorate(func):
        logname = name if name else func.__module__
        log = logging.getLogger(logname)
        logmsg = message if message else func.__name__

        @wraps(func)
        def wrapper(*args, **kwargs):
            log.log(level, logmsg)
            return func(*args, **kwargs)
```

```

    # Attach setter functions
    @attach_wrapper(wrapper)
    def set_level(newlevel):
        nonlocal level
        level = newlevel

    @attach_wrapper(wrapper)
    def set_message(newmsg):
        nonlocal logmsg
        logmsg = newmsg

    return wrapper
return decorate

# Example use
@logged(logging.DEBUG)
def add(x, y):
    return x + y

@logged(logging.CRITICAL, 'example')
def spam():
    print('Spam!')

```

Here is an interactive session that shows the various attributes being changed after definition:

```

>>> import logging
>>> logging.basicConfig(level=logging.DEBUG)
>>> add(2, 3)
DEBUG:__main__:add
5

>>> # Change the log message
>>> add.set_message('Add called')
>>> add(2, 3)
DEBUG:__main__:Add called
5

>>> # Change the log level
>>> add.set_level(logging.WARNING)
>>> add(2, 3)
WARNING:__main__:Add called
5
>>>

```

## Discussion

The key to this recipe lies in the accessor functions [e.g., `set_message()` and `set_level()`] that get attached to the wrapper as attributes. Each of these accessors allows internal parameters to be adjusted through the use of `nonlocal` assignments.

An amazing feature of this recipe is that the accessor functions will propagate through multiple levels of decoration (if all of your decorators utilize `@functools.wraps`). For example, suppose you introduced an additional decorator, such as the `@timethis` decorator from [Recipe 9.2](#), and wrote code like this:

```
@timethis
@logged(logging.DEBUG)
def countdown(n):
    while n > 0:
        n -= 1
```

You'll find that the accessor methods still work:

```
>>> countdown(10000000)
DEBUG:__main__:countdown
countdown 0.8198461532592773
>>> countdown.set_level(logging.WARNING)
>>> countdown.set_message("Counting down to zero")
>>> countdown(10000000)
WARNING:__main__:Counting down to zero
countdown 0.8225970268249512
>>>
```

You'll also find that it all still works exactly the same way if the decorators are composed in the opposite order, like this:

```
@logged(logging.DEBUG)
@timethis
def countdown(n):
    while n > 0:
        n -= 1
```

Although it's not shown, accessor functions to return the value of various settings could also be written just as easily by adding extra code such as this:

```
...
@attach_wrapper(wrapper)
def get_level():
    return level

# Alternative
wrapper.get_level = lambda: level
...
```

One extremely subtle facet of this recipe is the choice to use accessor functions in the first place. For example, you might consider an alternative formulation solely based on direct access to function attributes like this:

```
...
@wraps(func)
def wrapper(*args, **kwargs):
    wrapper.log.log(wrapper.level, wrapper.logmsg)
    return func(*args, **kwargs)

# Attach adjustable attributes
wrapper.level = level
wrapper.logmsg = logmsg
wrapper.log = log
...
```

This approach would work to a point, but only if it was the topmost decorator. If you had another decorator applied on top (such as the `@timethis` example), it would shadow the underlying attributes and make them unavailable for modification. The use of accessor functions avoids this limitation.

Last, but not least, the solution shown in this recipe might be a possible alternative for decorators defined as classes, as shown in [Recipe 9.9](#).

## 9.6. Defining a Decorator That Takes an Optional Argument

### Problem

You would like to write a single decorator that can be used without arguments, such as `@decorator`, or with optional arguments, such as `@decorator(x,y,z)`. However, there seems to be no straightforward way to do it due to differences in calling conventions between simple decorators and decorators taking arguments.

### Solution

Here is a variant of the logging code shown in [Recipe 9.5](#) that defines such a decorator:

```
from functools import wraps, partial
import logging

def logged(func=None, *, level=logging.DEBUG, name=None, message=None):
    if func is None:
        return partial(logged, level=level, name=name, message=message)

    logname = name if name else func.__module__
    log = logging.getLogger(logname)
    logmsg = message if message else func.__name__
```

```

@wraps(func)
def wrapper(*args, **kwargs):
    log.log(level, logmsg)
    return func(*args, **kwargs)
return wrapper

# Example use
@logged
def add(x, y):
    return x + y

@logged(level=logging.CRITICAL, name='example')
def spam():
    print('Spam!')

```

As you can see from the example, the decorator can be used in both a simple form (i.e., `@logged`) or with optional arguments supplied (i.e., `@logged(level=logging.CRITICAL, name='example')`).

## Discussion

The problem addressed by this recipe is really one of programming consistency. When using decorators, most programmers are used to applying them without any arguments at all or with arguments, as shown in the example. Technically speaking, a decorator where all arguments are optional could be applied, like this:

```

@logged()
def add(x, y):
    return x+y

```

However, this is not a form that's especially common, and might lead to common usage errors if programmers forget to add the extra parentheses. The recipe simply makes the decorator work with or without parentheses in a consistent way.

To understand how the code works, you need to have a firm understanding of how decorators get applied to functions and their calling conventions. For a simple decorator such as this:

```

# Example use
@logged
def add(x, y):
    return x + y

```

The calling sequence is as follows:

```

def add(x, y):
    return x + y
add = logged(add)

```

In this case, the function to be wrapped is simply passed to `logged` as the first argument. Thus, in the solution, the first argument of `logged()` is the function being wrapped. All of the other arguments must have default values.

For a decorator taking arguments such as this:

```
@logged(level=logging.CRITICAL, name='example')
def spam():
    print('Spam!')
```

The calling sequence is as follows:

```
def spam():
    print('Spam!')
spam = logged(level=logging.CRITICAL, name='example')(spam)
```

On the initial invocation of `logged()`, the function to be wrapped is not passed. Thus, in the decorator, it has to be optional. This, in turn, forces the other arguments to be specified by keyword. Furthermore, when arguments are passed, a decorator is supposed to return a function that accepts the function and wraps it (see [Recipe 9.5](#)). To do this, the solution uses a clever trick involving `functools.partial`. Specifically, it simply returns a partially applied version of itself where all arguments are fixed except for the function to be wrapped. See [Recipe 7.8](#) for more details about using `partial()`.

## 9.7. Enforcing Type Checking on a Function Using a Decorator

### Problem

You want to optionally enforce type checking of function arguments as a kind of assertion or contract.

### Solution

Before showing the solution code, the aim of this recipe is to have a means of enforcing type contracts on the input arguments to a function. Here is a short example that illustrates the idea:

```
>>> @typeassert(int, int)
... def add(x, y):
...     return x + y
...
>>>
>>> add(2, 3)
5
>>> add(2, 'hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```

File "contract.py", line 33, in wrapper
TypeError: Argument y must be <class 'int'>
>>>

```

Now, here is an implementation of the `@typeassert` decorator:

```

from inspect import signature
from functools import wraps

def typeassert(*ty_args, **ty_kwargs):
    def decorate(func):
        # If in optimized mode, disable type checking
        if not __debug__:
            return func

        # Map function argument names to supplied types
        sig = signature(func)
        bound_types = sig.bind_partial(*ty_args, **ty_kwargs).arguments

        @wraps(func)
        def wrapper(*args, **kwargs):
            bound_values = sig.bind(*args, **kwargs)
            # Enforce type assertions across supplied arguments
            for name, value in bound_values.arguments.items():
                if name in bound_types:
                    if not isinstance(value, bound_types[name]):
                        raise TypeError(
                            'Argument {} must be {}'.format(name, bound_types[name])
                        )
            return func(*args, **kwargs)
        return wrapper
    return decorate

```

You will find that this decorator is rather flexible, allowing types to be specified for all or a subset of a function's arguments. Moreover, types can be specified by position or by keyword. Here is an example:

```

>>> @typeassert(int, z=int)
... def spam(x, y, z=42):
...     print(x, y, z)
...
>>> spam(1, 2, 3)
1 2 3
>>> spam(1, 'hello', 3)
1 hello 3
>>> spam(1, 'hello', 'world')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "contract.py", line 33, in wrapper
TypeError: Argument z must be <class 'int'>
>>>

```



## Discussion

This recipe is an advanced decorator example that introduces a number of important and useful concepts.

First, one aspect of decorators is that they only get applied once, at the time of function definition. In certain cases, you may want to disable the functionality added by a decorator. To do this, simply have your decorator function return the function unwrapped. In the solution, the following code fragment returns the function unmodified if the value of the global `__debug__` variable is set to `False` (as is the case when Python executes in optimized mode with the `-O` or `-OO` options to the interpreter):

```
...
def decorate(func):
    # If in optimized mode, disable type checking
    if not __debug__:
        return func
    ...
```

Next, a tricky part of writing this decorator is that it involves examining and working with the argument signature of the function being wrapped. Your tool of choice here should be the `inspect.signature()` function. Simply stated, it allows you to extract signature information from a callable. For example:

```
>>> from inspect import signature
>>> def spam(x, y, z=42):
...     pass
...
>>> sig = signature(spam)
>>> print(sig)
(x, y, z=42)
>>> sig.parameters
mappingproxy(OrderedDict([('x', <Parameter at 0x10077a050 'x'>),
('y', <Parameter at 0x10077a158 'y'>), ('z', <Parameter at 0x10077a1b0 'z'>)]))
>>> sig.parameters['z'].name
'z'
>>> sig.parameters['z'].default
42
>>> sig.parameters['z'].kind
<_ParameterKind: 'POSITIONAL_OR_KEYWORD'>
>>>
```

In the first part of our decorator, we use the `bind_partial()` method of signatures to perform a partial binding of the supplied types to argument names. Here is an example of what happens:

```
>>> bound_types = sig.bind_partial(int, z=int)
>>> bound_types
<inspect.BoundArguments object at 0x10069bb50>
>>> bound_types.arguments
```

```
OrderedDict([('x', <class 'int'>), ('z', <class 'int'>)])
>>>
```

In this partial binding, you will notice that missing arguments are simply ignored (i.e., there is no binding for argument `y`). However, the most important part of the binding is the creation of the ordered dictionary `bound_types.arguments`. This dictionary maps the argument names to the supplied values in the same order as the function signature. In the case of our decorator, this mapping contains the type assertions that we’re going to enforce.

In the actual wrapper function made by the decorator, the `sig.bind()` method is used. `bind()` is like `bind_partial()` except that it does not allow for missing arguments. So, here is what happens:

```
>>> bound_values = sig.bind(1, 2, 3)
>>> bound_values.arguments
OrderedDict([('x', 1), ('y', 2), ('z', 3)])
>>>
```

Using this mapping, it is relatively easy to enforce the required assertions.

```
>>> for name, value in bound_values.arguments.items():
...     if name in bound_types.arguments:
...         if not isinstance(value, bound_types.arguments[name]):
...             raise TypeError()
...
>>>
```

A somewhat subtle aspect of the solution is that the assertions do not get applied to unsupplied arguments with default values. For example, this code works, even though the default value of `items` is of the “wrong” type:

```
>>> @typeassert(int, list)
... def bar(x, items=None):
...     if items is None:
...         items = []
...     items.append(x)
...     return items
>>> bar(2)
[2]
>>> bar(2,3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "contract.py", line 33, in wrapper
TypeError: Argument items must be <class 'list'>
>>> bar(4, [1, 2, 3])
[1, 2, 3, 4]
>>>
```

A final point of design discussion might be the use of decorator arguments versus function annotations. For example, why not write the decorator to look at annotations like this?

```
@typeassert
def spam(x:int, y, z:int = 42):
    print(x,y,z)
```

One possible reason for not using annotations is that each argument to a function can only have a single annotation assigned. Thus, if the annotations are used for type assertions, they can't really be used for anything else. Likewise, the `@typeassert` decorator won't work with functions that use annotations for a different purpose. By using decorator arguments, as shown in the solution, the decorator becomes a lot more general purpose and can be used with any function whatsoever—even functions that use annotations.

More information about function signature objects can be found in [PEP 362](#), as well as the [documentation for the inspect module](#). [Recipe 9.16](#) also has an additional example.

## 9.8. Defining Decorators As Part of a Class

### Problem

You want to define a decorator inside a class definition and apply it to other functions or methods.

### Solution

Defining a decorator inside a class is straightforward, but you first need to sort out the manner in which the decorator will be applied. Specifically, whether it is applied as an instance or a class method. Here is an example that illustrates the difference:

```
from functools import wraps

class A:
    # Decorator as an instance method
    def decorator1(self, func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            print('Decorator 1')
            return func(*args, **kwargs)
        return wrapper

    # Decorator as a class method
    @classmethod
    def decorator2(cls, func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            print('Decorator 2')
            return func(*args, **kwargs)
        return wrapper
```

Here is an example of how the two decorators would be applied:

```

# As an instance method
a = A()

@a.decorator1
def spam():
    pass

# As a class method
@A.decorator2
def grok():
    pass

```

If you look carefully, you'll notice that one is applied from an instance `a` and the other is applied from the class `A`.

## Discussion

Defining decorators in a class might look odd at first glance, but there are examples of this in the standard library. In particular, the built-in `@property` decorator is actually a class with `getter()`, `setter()`, and `deleter()` methods that each act as a decorator. For example:

```

class Person:
    # Create a property instance
    first_name = property()

    # Apply decorator methods
    @first_name.getter
    def first_name(self):
        return self._first_name

    @first_name.setter
    def first_name(self, value):
        if not isinstance(value, str):
            raise TypeError('Expected a string')
        self._first_name = value

```

The key reason why it's defined in this way is that the various decorator methods are manipulating state on the associated property instance. So, if you ever had a problem where decorators needed to record or combine information behind the scenes, it's a sensible approach.

A common confusion when writing decorators in classes is getting tripped up by the proper use of the extra `self` or `cls` arguments in the decorator code itself. Although the outermost decorator function, such as `decorator1()` or `decorator2()`, needs to provide a `self` or `cls` argument (since they're part of a class), the wrapper function created inside doesn't generally need to include an extra argument. This is why the `wrapper()` function created in both decorators doesn't include a `self` argument. The only time you would ever need this argument is in situations where you actually needed

to access parts of an instance in the wrapper. Otherwise, you just don't have to worry about it.

A final subtle facet of having decorators defined in a class concerns their potential use with inheritance. For example, suppose you want to apply one of the decorators defined in class A to methods defined in a subclass B. To do that, you would need to write code like this:

```
class B(A):
    @A.decorator2
    def bar(self):
        pass
```

In particular, the decorator in question has to be defined as a class method and you have to explicitly use the name of the superclass A when applying it. You can't use a name such as `@B.decorator2`, because at the time of method definition, class B has not yet been created.

## 9.9. Defining Decorators As Classes

### Problem

You want to wrap functions with a decorator, but the result is going to be a callable instance. You need your decorator to work both inside and outside class definitions.

### Solution

To define a decorator as an instance, you need to make sure it implements the `__call__()` and `__get__()` methods. For example, this code defines a class that puts a simple profiling layer around another function:

```
import types
from functools import wraps

class Profiled:
    def __init__(self, func):
        wraps(func)(self)
        self.ncalls = 0

    def __call__(self, *args, **kwargs):
        self.ncalls += 1
        return self.__wrapped__(*args, **kwargs)

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return types.MethodType(self, instance)
```

To use this class, you use it like a normal decorator, either inside or outside of a class:

```
@Profiled
def add(x, y):
    return x + y

class Spam:
    @Profiled
    def bar(self, x):
        print(self, x)
```

Here is an interactive session that shows how these functions work:

```
>>> add(2, 3)
5
>>> add(4, 5)
9
>>> add.ncalls
2
>>> s = Spam()
>>> s.bar(1)
<__main__.Spam object at 0x10069e9d0> 1
>>> s.bar(2)
<__main__.Spam object at 0x10069e9d0> 2
>>> s.bar(3)
<__main__.Spam object at 0x10069e9d0> 3
>>> Spam.bar.ncalls
3
```

## Discussion

Defining a decorator as a class is usually straightforward. However, there are some rather subtle details that deserve more explanation, especially if you plan to apply the decorator to instance methods.

First, the use of the `functools.wraps()` function serves the same purpose here as it does in normal decorators—namely to copy important metadata from the wrapped function to the callable instance.

Second, it is common to overlook the `__get__()` method shown in the solution. If you omit the `__get__()` and keep all of the other code the same, you'll find that bizarre things happen when you try to invoke decorated instance methods. For example:

```
>>> s = Spam()
>>> s.bar(3)
Traceback (most recent call last):
...
TypeError: spam() missing 1 required positional argument: 'x'
```

The reason it breaks is that whenever functions implementing methods are looked up in a class, their `__get__()` method is invoked as part of the descriptor protocol, which

is described in [Recipe 8.9](#). In this case, the purpose of `__get__()` is to create a bound method object (which ultimately supplies the `self` argument to the method). Here is an example that illustrates the underlying mechanics:

```
>>> s = Spam()
>>> def grok(self, x):
...     pass
...
>>> grok.__get__(s, Spam)
<bound method Spam.grok of <__main__.Spam object at 0x100671e90>>
>>>
```

In this recipe, the `__get__()` method is there to make sure bound method objects get created properly. `type.MethodType()` creates a bound method manually for use here. Bound methods only get created if an instance is being used. If the method is accessed on a class, the instance argument to `__get__()` is set to `None` and the `Profiled` instance itself is just returned. This makes it possible for someone to extract its `ncalls` attribute, as shown.

If you want to avoid some of this of this mess, you might consider an alternative formulation of the decorator using closures and `nonlocal` variables, as described in [Recipe 9.5](#). For example:

```
import types
from functools import wraps

def profiled(func):
    ncalls = 0
    @wraps(func)
    def wrapper(*args, **kwargs):
        nonlocal ncalls
        ncalls += 1
        return func(*args, **kwargs)
    wrapper.ncalls = lambda: ncalls
    return wrapper

# Example
@profiled
def add(x, y):
    return x + y
```

This example almost works in exactly the same way except that access to `ncalls` is now provided through a function attached as a function attribute. For example:

```
>>> add(2, 3)
5
>>> add(4, 5)
9
>>> add.ncalls()
2
>>>
```

## 9.10. Applying Decorators to Class and Static Methods

### Problem

You want to apply a decorator to a class or static method.

### Solution

Applying decorators to class and static methods is straightforward, but make sure that your decorators are applied before `@classmethod` or `@staticmethod`. For example:

```
import time
from functools import wraps

# A simple decorator
def timethis(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        r = func(*args, **kwargs)
        end = time.time()
        print(end-start)
        return r
    return wrapper

# Class illustrating application of the decorator to different kinds of methods
class Spam:
    @timethis
    def instance_method(self, n):
        print(self, n)
        while n > 0:
            n -= 1

    @classmethod
    @timethis
    def class_method(cls, n):
        print(cls, n)
        while n > 0:
            n -= 1

    @staticmethod
    @timethis
    def static_method(n):
        print(n)
        while n > 0:
            n -= 1
```

The resulting class and static methods should operate normally, but have the extra timing:



```

>>> s = Spam()
>>> s.instance_method(1000000)
<__main__.Spam object at 0x1006a6050> 1000000
0.11817407608032227
>>> Spam.class_method(1000000)
<class '__main__.Spam'> 1000000
0.11334395408630371
>>> Spam.static_method(1000000)
1000000
0.11740279197692871
>>>

```

## Discussion

If you get the order of decorators wrong, you'll get an error. For example, if you use the following:

```

class Spam:
    ...
    @timethis
    @staticmethod
    def static_method(n):
        print(n)
        while n > 0:
            n -= 1

```

Then the static method will crash:

```

>>> Spam.static_method(1000000)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "timethis.py", line 6, in wrapper
    start = time.time()
TypeError: 'staticmethod' object is not callable
>>>

```

The problem here is that `@classmethod` and `@staticmethod` don't actually create objects that are directly callable. Instead, they create special descriptor objects, as described in [Recipe 8.9](#). Thus, if you try to use them like functions in another decorator, the decorator will crash. Making sure that these decorators appear first in the decorator list fixes the problem.

One situation where this recipe is of critical importance is in defining class and static methods in abstract base classes, as described in [Recipe 8.12](#). For example, if you want to define an abstract class method, you can use this code:

```

from abc import ABCMeta, abstractmethod

class A(metaclass=ABCMeta):
    @classmethod
    @abstractmethod

```

```
def method(cls):  
    pass
```

In this code, the order of `@classmethod` and `@abstractmethod` matters. If you flip the two decorators around, everything breaks.

## 9.11. Writing Decorators That Add Arguments to Wrapped Functions

### Problem

You want to write a decorator that adds an extra argument to the calling signature of the wrapped function. However, the added argument can't interfere with the existing calling conventions of the function.

### Solution

Extra arguments can be injected into the calling signature using keyword-only arguments. Consider the following decorator:

```
from functools import wraps  
  
def optional_debug(func):  
    @wraps(func)  
    def wrapper(*args, debug=False, **kwargs):  
        if debug:  
            print('Calling', func.__name__)  
        return func(*args, **kwargs)  
    return wrapper
```

Here is an example of how the decorator works:

```
>>> @optional_debug  
... def spam(a,b,c):  
...     print(a,b,c)  
...  
>>> spam(1,2,3)  
1 2 3  
>>> spam(1,2,3, debug=True)  
Calling spam  
1 2 3  
>>>
```

### Discussion

Adding arguments to the signature of wrapped functions is not the most common example of using decorators. However, it might be a useful technique in avoiding certain kinds of code replication patterns. For example, if you have code like this:

```

def a(x, debug=False):
    if debug:
        print('Calling a')
    ...

def b(x, y, z, debug=False):
    if debug:
        print('Calling b')
    ...

def c(x, y, debug=False):
    if debug:
        print('Calling c')
    ...

```

You can refactor it into the following:

```

@optional_debug
def a(x):
    ...

@optional_debug
def b(x, y, z):
    ...

@optional_debug
def c(x, y):
    ...

```

The implementation of this recipe relies on the fact that keyword-only arguments are easy to add to functions that also accept `*args` and `**kwargs` parameters. By using a keyword-only argument, it gets singled out as a special case and removed from subsequent calls that only use the remaining positional and keyword arguments.

One tricky part here concerns a potential name clash between the added argument and the arguments of the function being wrapped. For example, if the `@optional_debug` decorator was applied to a function that already had a `debug` argument, then it would break. If that's a concern, an extra check could be added:

```

from functools import wraps
import inspect

def optional_debug(func):
    if 'debug' in inspect.getargspec(func).args:
        raise TypeError('debug argument already defined')

    @wraps(func)
    def wrapper(*args, debug=False, **kwargs):
        if debug:
            print('Calling', func.__name__)
        return func(*args, **kwargs)
    return wrapper

```

A final refinement to this recipe concerns the proper management of function signatures. An astute programmer will realize that the signature of wrapped functions is wrong. For example:

```
>>> @optional_debug
... def add(x,y):
...     return x+y
...
>>> import inspect
>>> print(inspect.signature(add))
(x, y)
>>>
```

This can be fixed by making the following modification:

```
from functools import wraps
import inspect

def optional_debug(func):
    if 'debug' in inspect.getargspec(func).args:
        raise TypeError('debug argument already defined')

    @wraps(func)
    def wrapper(*args, debug=False, **kwargs):
        if debug:
            print('Calling', func.__name__)
        return func(*args, **kwargs)

    sig = inspect.signature(func)
    parms = list(sig.parameters.values())
    parms.append(inspect.Parameter('debug',
                                   inspect.Parameter.KEYWORD_ONLY,
                                   default=False))
    wrapper.__signature__ = sig.replace(parameters=parms)
    return wrapper
```

With this change, the signature of the wrapper will now correctly reflect the presence of the debug argument. For example:

```
>>> @optional_debug
... def add(x,y):
...     return x+y
...
>>> print(inspect.signature(add))
(x, y, *, debug=False)
>>> add(2,3)
5
>>>
```

See [Recipe 9.16](#) for more information about function signatures.

## 9.12. Using Decorators to Patch Class Definitions

### Problem

You want to inspect or rewrite portions of a class definition to alter its behavior, but without using inheritance or metaclasses.

### Solution

This might be a perfect use for a class decorator. For example, here is a class decorator that rewrites the `__getattrattribute__` special method to perform logging.

```
def log_getattribute(cls):
    # Get the original implementation
    orig_getattribute = cls.__getattrattribute__

    # Make a new definition
    def new_getattribute(self, name):
        print('getting:', name)
        return orig_getattribute(self, name)

    # Attach to the class and return
    cls.__getattrattribute__ = new_getattribute
    return cls

# Example use
@log_getattribute
class A:
    def __init__(self, x):
        self.x = x
    def spam(self):
        pass
```

Here is what happens if you try to use the class in the solution:

```
>>> a = A(42)
>>> a.x
getting: x
42
>>> a.spam()
getting: spam
>>>
```

### Discussion

Class decorators can often be used as a straightforward alternative to other more advanced techniques involving mixins or metaclasses. For example, an alternative implementation of the solution might involve inheritance, as in the following:

```

class LoggedGetattribute:
    def __getattribute__(self, name):
        print('getting:', name)
        return super().__getattribute__(name)

# Example:
class A(LoggedGetattribute):
    def __init__(self, x):
        self.x = x
    def spam(self):
        pass

```

This works, but to understand it, you have to have some awareness of the method resolution order, `super()`, and other aspects of inheritance, as described in [Recipe 8.7](#). In some sense, the class decorator solution is much more direct in how it operates, and it doesn't introduce new dependencies into the inheritance hierarchy. As it turns out, it's also just a bit faster, due to not relying on the `super()` function.

If you are applying multiple class decorators to a class, the application order might matter. For example, a decorator that replaces a method with an entirely new implementation would probably need to be applied before a decorator that simply wraps an existing method with some extra logic.

See [Recipe 8.13](#) for another example of class decorators in action.

## 9.13. Using a Metaclass to Control Instance Creation

### Problem

You want to change the way in which instances are created in order to implement singletons, caching, or other similar features.

### Solution

As Python programmers know, if you define a class, you call it like a function to create instances. For example:

```

class Spam:
    def __init__(self, name):
        self.name = name

a = Spam('Guido')
b = Spam('Diana')

```

If you want to customize this step, you can do it by defining a metaclass and reimplementing its `__call__()` method in some way. To illustrate, suppose that you didn't want anyone creating instances at all:

```

class NoInstances(type):
    def __call__(self, *args, **kwargs):
        raise TypeError("Can't instantiate directly")

# Example
class Spam(metaclass=NoInstances):
    @staticmethod
    def grok(x):
        print('Spam.grok')

```

In this case, users can call the defined static method, but it's impossible to create an instance in the normal way. For example:

```

>>> Spam.grok(42)
Spam.grok
>>> s = Spam()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "example1.py", line 7, in __call__
    raise TypeError("Can't instantiate directly")
TypeError: Can't instantiate directly
>>>

```

Now, suppose you want to implement the singleton pattern (i.e., a class where only one instance is ever created). That is also relatively straightforward, as shown here:

```

class Singleton(type):
    def __init__(self, *args, **kwargs):
        self.__instance = None
        super().__init__(*args, **kwargs)

    def __call__(self, *args, **kwargs):
        if self.__instance is None:
            self.__instance = super().__call__(*args, **kwargs)
            return self.__instance
        else:
            return self.__instance

# Example
class Spam(metaclass=Singleton):
    def __init__(self):
        print('Creating Spam')

```

In this case, only one instance ever gets created. For example:

```

>>> a = Spam()
Creating Spam
>>> b = Spam()
>>> a is b
True
>>> c = Spam()
>>> a is c
True
>>>

```

Finally, suppose you want to create cached instances, as described in [Recipe 8.25](#). Here's a metaclass that implements it:

```
import weakref

class Cached(type):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.__cache = weakref.WeakValueDictionary()

    def __call__(self, *args):
        if args in self.__cache:
            return self.__cache[args]
        else:
            obj = super().__call__(*args)
            self.__cache[args] = obj
            return obj

# Example
class Spam(metaclass=Cached):
    def __init__(self, name):
        print('Creating Spam({!r})'.format(name))
        self.name = name
```

Here's an example showing the behavior of this class:

```
>>> a = Spam('Guido')
Creating Spam('Guido')
>>> b = Spam('Diana')
Creating Spam('Diana')
>>> c = Spam('Guido')           # Cached
>>> a is b
False
>>> a is c                       # Cached value returned
True
>>>
```

## Discussion

Using a metaclass to implement various instance creation patterns can often be a much more elegant approach than other solutions not involving metaclasses. For example, if you didn't use a metaclass, you might have to hide the classes behind some kind of extra factory function. For example, to get a singleton, you might use a hack such as the following:

```
class _Spam:
    def __init__(self):
        print('Creating Spam')

_spam_instance = None
def Spam():
    global _spam_instance
```



```

if _spam_instance is not None:
    return _spam_instance
else:
    _spam_instance = _Spam()
    return _spam_instance

```

Although the solution involving metaclasses involves a much more advanced concept, the resulting code feels cleaner and less hacked together.

See [Recipe 8.25](#) for more information on creating cached instances, weak references, and other details.

## 9.14. Capturing Class Attribute Definition Order

### Problem

You want to automatically record the order in which attributes and methods are defined inside a class body so that you can use it in various operations (e.g., serializing, mapping to databases, etc.).

### Solution

Capturing information about the body of class definition is easily accomplished through the use of a metaclass. Here is an example of a metaclass that uses an `OrderedDict` to capture definition order of descriptors:

```

from collections import OrderedDict

# A set of descriptors for various types
class Typed:
    _expected_type = type(None)
    def __init__(self, name=None):
        self._name = name

    def __set__(self, instance, value):
        if not isinstance(value, self._expected_type):
            raise TypeError('Expected ' + str(self._expected_type))
        instance.__dict__[self._name] = value

class Integer(Typed):
    _expected_type = int

class Float(Typed):
    _expected_type = float

class String(Typed):
    _expected_type = str

# Metaclass that uses an OrderedDict for class body

```

```

class OrderedMeta(type):
    def __new__(cls, clsname, bases, clsdict):
        d = dict(clsdict)
        order = []
        for name, value in clsdict.items():
            if isinstance(value, Typed):
                value._name = name
                order.append(name)
        d['_order'] = order
        return type.__new__(cls, clsname, bases, d)

    @classmethod
    def __prepare__(cls, clsname, bases):
        return OrderedDict()

```

In this metaclass, the definition order of descriptors is captured by using an Ordered Dict during the execution of the class body. The resulting order of names is then extracted from the dictionary and stored into a class attribute `_order`. This can then be used by methods of the class in various ways. For example, here is a simple class that uses the ordering to implement a method for serializing the instance data as a line of CSV data:

```

class Structure(metaclass=OrderedMeta):
    def as_csv(self):
        return ','.join(str(getattr(self,name)) for name in self._order)

# Example use
class Stock(Structure):
    name = String()
    shares = Integer()
    price = Float()
    def __init__(self, name, shares, price):
        self.name = name
        self.shares = shares
        self.price = price

```

Here is an interactive session illustrating the use of the Stock class in the example:

```

>>> s = Stock('GOOG',100,490.1)
>>> s.name
'GOOG'
>>> s.as_csv()
'GOOG,100,490.1'
>>> t = Stock('AAPL', 'a lot', 610.23)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "dupmethod.py", line 34, in __init__
TypeError: shares expects <class 'int'>
>>>

```

## Discussion

The entire key to this recipe is the `__prepare__()` method, which is defined in the `OrderedMeta` metaclass. This method is invoked immediately at the start of a class definition with the class name and base classes. It must then return a mapping object to use when processing the class body. By returning an `OrderedDict` instead of a normal dictionary, the resulting definition order is easily captured.

It is possible to extend this functionality even further if you are willing to make your own dictionary-like objects. For example, consider this variant of the solution that rejects duplicate definitions:

```
from collections import OrderedDict

class NoDupOrderedDict(OrderedDict):
    def __init__(self, clsname):
        self.clsname = clsname
        super().__init__()
    def __setitem__(self, name, value):
        if name in self:
            raise TypeError('{} already defined in {}'.format(name, self.clsname))
        super().__setitem__(name, value)

class OrderedMeta(type):
    def __new__(cls, clsname, bases, clsdict):
        d = dict(clsdict)
        d['_order'] = [name for name in clsdict if name[0] != '_']
        return type.__new__(cls, clsname, bases, d)

    @classmethod
    def __prepare__(cls, clsname, bases):
        return NoDupOrderedDict(clsname)
```

Here's what happens if you use this metaclass and make a class with duplicate entries:

```
>>> class A(metaclass=OrderedMeta):
...     def spam(self):
...         pass
...     def spam(self):
...         pass
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in A
  File "dupmethod2.py", line 25, in __setitem__
    (name, self.clsname))
TypeError: spam already defined in A
>>>
```

A final important part of this recipe concerns the treatment of the modified dictionary in the metaclass `__new__()` method. Even though the class was defined using an alter-

native dictionary, you still have to convert this dictionary to a proper `dict` instance when making the final class object. This is the purpose of the `d = dict(clsdict)` statement.

Being able to capture definition order is a subtle but important feature for certain kinds of applications. For instance, in an object relational mapper, classes might be written in a manner similar to that shown in the example:

```
class Stock(Model):
    name = String()
    shares = Integer()
    price = Float()
```

Underneath the covers, the code might want to capture the definition order to map objects to tuples or rows in a database table (e.g., similar to the functionality of the `as_csv()` method in the example). The solution shown is very straightforward and often simpler than alternative approaches (which typically involve maintaining hidden counters within the descriptor classes).

## 9.15. Defining a Metaclass That Takes Optional Arguments

### Problem

You want to define a metaclass that allows class definitions to supply optional arguments, possibly to control or configure aspects of processing during type creation.

### Solution

When defining classes, Python allows a metaclass to be specified using the `metaclass` keyword argument in the `class` statement. For example, with abstract base classes:

```
from abc import ABCMeta, abstractmethod

class IStream(metaclass=ABCMeta):
    @abstractmethod
    def read(self, maxsize=None):
        pass

    @abstractmethod
    def write(self, data):
        pass
```

However, in custom metaclasses, additional keyword arguments can be supplied, like this:

```
class Spam(metaclass=MyMeta, debug=True, synchronize=True):
    ...
```

To support such keyword arguments in a metaclass, make sure you define them on the `__prepare__()`, `__new__()`, and `__init__()` methods using keyword-only arguments, like this:

```
class MyMeta(type):
    # Optional
    @classmethod
    def __prepare__(cls, name, bases, *, debug=False, synchronize=False):
        # Custom processing
        ...
        return super().__prepare__(name, bases)

    # Required
    def __new__(cls, name, bases, ns, *, debug=False, synchronize=False):
        # Custom processing
        ...
        return super().__new__(cls, name, bases, ns)

    # Required
    def __init__(self, name, bases, ns, *, debug=False, synchronize=False):
        # Custom processing
        ...
        super().__init__(name, bases, ns)
```

## Discussion

Adding optional keyword arguments to a metaclass requires that you understand all of the steps involved in class creation, because the extra arguments are passed to every method involved. The `__prepare__()` method is called first and used to create the class namespace prior to the body of any class definition being processed. Normally, this method simply returns a dictionary or other mapping object. The `__new__()` method is used to instantiate the resulting type object. It is called after the class body has been fully executed. The `__init__()` method is called last and used to perform any additional initialization steps.

When writing metaclasses, it is somewhat common to only define a `__new__()` or `__init__()` method, but not both. However, if extra keyword arguments are going to be accepted, then both methods must be provided and given compatible signatures. The default `__prepare__()` method accepts any set of keyword arguments, but ignores them. You only need to define it yourself if the extra arguments would somehow affect management of the class namespace creation.

The use of keyword-only arguments in this recipe reflects the fact that such arguments will only be supplied by keyword during class creation.

The specification of keyword arguments to configure a metaclass might be viewed as an alternative to using class variables for a similar purpose. For example:

```

class Spam(metaclass=MyMeta):
    debug = True
    synchronize = True
    ...

```

The advantage to supplying such parameters as an argument is that they don't pollute the class namespace with extra names that only pertain to class creation and not the subsequent execution of statements in the class. In addition, they are available to the `__prepare__()` method, which runs prior to processing any statements in the class body. Class variables, on the other hand, would only be accessible in the `__new__()` and `__init__()` methods of a metaclass.

## 9.16. Enforcing an Argument Signature on `*args` and `**kwargs`

### Problem

You've written a function or method that uses `*args` and `**kwargs`, so that it can be general purpose, but you would also like to check the passed arguments to see if they match a specific function calling signature.

### Solution

For any problem where you want to manipulate function calling signatures, you should use the signature features found in the `inspect` module. Two classes, `Signature` and `Parameter`, are of particular interest here. Here is an interactive example of creating a function signature:

```

>>> from inspect import Signature, Parameter
>>> # Make a signature for a func(x, y=42, *, z=None)
>>> parms = [ Parameter('x', Parameter.POSITIONAL_OR_KEYWORD),
...           Parameter('y', Parameter.POSITIONAL_OR_KEYWORD, default=42),
...           Parameter('z', Parameter.KEYWORD_ONLY, default=None) ]
>>> sig = Signature(parms)
>>> print(sig)
(x, y=42, *, z=None)
>>>

```

Once you have a signature object, you can easily bind it to `*args` and `**kwargs` using the signature's `bind()` method, as shown in this simple example:

```

>>> def func(*args, **kwargs):
...     bound_values = sig.bind(*args, **kwargs)
...     for name, value in bound_values.arguments.items():
...         print(name, value)
...
>>> # Try various examples
>>> func(1, 2, z=3)

```

```

x 1
y 2
z 3
>>> func(1)
x 1
>>> func(1, z=3)
x 1
z 3
>>> func(y=2, x=1)
x 1
y 2
>>> func(1, 2, 3, 4)
Traceback (most recent call last):
...
  File "/usr/local/lib/python3.3/inspect.py", line 1972, in _bind
    raise TypeError('too many positional arguments')
TypeError: too many positional arguments
>>> func(y=2)
Traceback (most recent call last):
...
  File "/usr/local/lib/python3.3/inspect.py", line 1961, in _bind
    raise TypeError(msg) from None
TypeError: 'x' parameter lacking default value
>>> func(1, y=2, x=3)
Traceback (most recent call last):
...
  File "/usr/local/lib/python3.3/inspect.py", line 1985, in _bind
    '{arg!r}'.format(arg=param.name))
TypeError: multiple values for argument 'x'
>>>

```

As you can see, the binding of a signature to the passed arguments enforces all of the usual function calling rules concerning required arguments, defaults, duplicates, and so forth.

Here is a more concrete example of enforcing function signatures. In this code, a base class has defined an extremely general-purpose version of `__init__()`, but subclasses are expected to supply an expected signature.

```

from inspect import Signature, Parameter

def make_sig(*names):
    parms = [Parameter(name, Parameter.POSITIONAL_OR_KEYWORD)
              for name in names]
    return Signature(parms)

class Structure:
    __signature__ = make_sig()
    def __init__(self, *args, **kwargs):
        bound_values = self.__signature__.bind(*args, **kwargs)
        for name, value in bound_values.arguments.items():
            setattr(self, name, value)

```

```
# Example use
class Stock(Structure):
    __signature__ = make_sig('name', 'shares', 'price')

class Point(Structure):
    __signature__ = make_sig('x', 'y')
```

Here is an example of how the Stock class works:

```
>>> import inspect
>>> print(inspect.signature(Stock))
(name, shares, price)
>>> s1 = Stock('ACME', 100, 490.1)
>>> s2 = Stock('ACME', 100)
Traceback (most recent call last):
...
TypeError: 'price' parameter lacking default value
>>> s3 = Stock('ACME', 100, 490.1, shares=50)
Traceback (most recent call last):
...
TypeError: multiple values for argument 'shares'
>>>
```

## Discussion

The use of functions involving `*args` and `**kwargs` is very common when trying to make general-purpose libraries, write decorators or implement proxies. However, one downside of such functions is that if you want to implement your own argument checking, it can quickly become an unwieldy mess. As an example, see [Recipe 8.11](#). The use of a signature object simplifies this.

In the last example of the solution, it might make sense to create signature objects through the use of a custom metaclass. Here is an alternative implementation that shows how to do this:

```
from inspect import Signature, Parameter

def make_sig(*names):
    parms = [Parameter(name, Parameter.POSITIONAL_OR_KEYWORD)
              for name in names]
    return Signature(parms)

class StructureMeta(type):
    def __new__(cls, clsname, bases, clsdict):
        clsdict['__signature__'] = make_sig(*clsdict.get('_fields', []))
        return super().__new__(cls, clsname, bases, clsdict)

class Structure(metaclass=StructureMeta):
    _fields = []
    def __init__(self, *args, **kwargs):
```



```

        bound_values = self.__signature__.bind(*args, **kwargs)
        for name, value in bound_values.arguments.items():
            setattr(self, name, value)

# Example
class Stock(Structure):
    _fields = ['name', 'shares', 'price']

class Point(Structure):
    _fields = ['x', 'y']

```

When defining custom signatures, it is often useful to store the signature in a special attribute `__signature__`, as shown. If you do this, code that uses the `inspect` module to perform introspection will see the signature and report it as the calling convention. For example:

```

>>> import inspect
>>> print(inspect.signature(Stock))
(name, shares, price)
>>> print(inspect.signature(Point))
(x, y)
>>>

```

## 9.17. Enforcing Coding Conventions in Classes

### Problem

Your program consists of a large class hierarchy and you would like to enforce certain kinds of coding conventions (or perform diagnostics) to help maintain programmer sanity.

### Solution

If you want to monitor the definition of classes, you can often do it by defining a metaclass. A basic metaclass is usually defined by inheriting from `type` and redefining its `__new__()` method or `__init__()` method. For example:

```

class MyMeta(type):
    def __new__(self, clsname, bases, clsdict):
        # clsname is name of class being defined
        # bases is tuple of base classes
        # clsdict is class dictionary
        return super().__new__(cls, clsname, bases, clsdict)

```

Alternatively, if `__init__()` is defined:

```

class MyMeta(type):
    def __init__(self, clsname, bases, clsdict):
        super().__init__(clsname, bases, clsdict)
        # clsname is name of class being defined

```

```
# bases is tuple of base classes  
# clsdict is class dictionary
```

To use a metaclass, you would generally incorporate it into a top-level base class from which other objects inherit. For example:

```
class Root(metaclass=MyMeta):  
    pass  
  
class A(Root):  
    pass  
  
class B(Root):  
    pass
```

A key feature of a metaclass is that it allows you to examine the contents of a class at the time of definition. Inside the redefined `__init__()` method, you are free to inspect the class dictionary, base classes, and more. Moreover, once a metaclass has been specified for a class, it gets inherited by all of the subclasses. Thus, a sneaky framework builder can specify a metaclass for one of the top-level classes in a large hierarchy and capture the definition of all classes under it.

As a concrete albeit whimsical example, here is a metaclass that rejects any class definition containing methods with mixed-case names (perhaps as a means for annoying Java programmers):

```
class NoMixedCaseMeta(type):  
    def __new__(cls, clsname, bases, clsdict):  
        for name in clsdict:  
            if name.lower() != name:  
                raise TypeError('Bad attribute name: ' + name)  
        return super().__new__(cls, clsname, bases, clsdict)  
  
class Root(metaclass=NoMixedCaseMeta):  
    pass  
  
class A(Root):  
    def foo_bar(self):        # Ok  
        pass  
  
class B(Root):  
    def fooBar(self):         # TypeError  
        pass
```

As a more advanced and useful example, here is a metaclass that checks the definition of redefined methods to make sure they have the same calling signature as the original method in the superclass.

```
from inspect import signature  
import logging  
  
class MatchSignaturesMeta(type):
```

```

def __init__(self, clsname, bases, clsdict):
    super().__init__(clsname, bases, clsdict)
    sup = super(self, self)
    for name, value in clsdict.items():
        if name.startswith('_') or not callable(value):
            continue
        # Get the previous definition (if any) and compare the signatures
        prev_dfn = getattr(sup, name, None)
        if prev_dfn:
            prev_sig = signature(prev_dfn)
            val_sig = signature(value)
            if prev_sig != val_sig:
                logging.warning('Signature mismatch in %s. %s != %s',
                                value.__qualname__, prev_sig, val_sig)

# Example
class Root(metaclass=MatchSignaturesMeta):
    pass

class A(Root):
    def foo(self, x, y):
        pass

    def spam(self, x, *, z):
        pass

# Class with redefined methods, but slightly different signatures
class B(A):
    def foo(self, a, b):
        pass

    def spam(self, x, z):
        pass

```

If you run this code, you will get output such as the following:

```

WARNING:root:Signature mismatch in B.spam. (self, x, *, z) != (self, x, z)
WARNING:root:Signature mismatch in B.foo. (self, x, y) != (self, a, b)

```

Such warnings might be useful in catching subtle program bugs. For example, code that relies on keyword argument passing to a method will break if a subclass changes the argument names.

## Discussion

In large object-oriented programs, it can sometimes be useful to put class definitions under the control of a metaclass. The metaclass can observe class definitions and be used to alert programmers to potential problems that might go unnoticed (e.g., using slightly incompatible method signatures).

One might argue that such errors would be better caught by program analysis tools or IDEs. To be sure, such tools are useful. However, if you're creating a framework or library that's going to be used by others, you often don't have any control over the rigor of their development practices. Thus, for certain kinds of applications, it might make sense to put a bit of extra checking in a metaclass if such checking would result in a better user experience.

The choice of redefining `__new__()` or `__init__()` in a metaclass depends on how you want to work with the resulting class. `__new__()` is invoked prior to class creation and is typically used when a metaclass wants to alter the class definition in some way (by changing the contents of the class dictionary). The `__init__()` method is invoked after a class has been created, and is useful if you want to write code that works with the fully formed class object. In the last example, this is essential since it is using the `super()` function to search for prior definitions. This only works once the class instance has been created and the underlying method resolution order has been set.

The last example also illustrates the use of Python's function signature objects. Essentially, the metaclass takes each callable definition in a class, searches for a prior definition (if any), and then simply compares their calling signatures using `inspect.signature()`.

Last, but not least, the line of code that uses `super(self, self)` is not a typo. When working with a metaclass, it's important to realize that the `self` is actually a class object. So, that statement is actually being used to find definitions located further up the class hierarchy that make up the parents of `self`.

## 9.18. Defining Classes Programmatically

### Problem

You're writing code that ultimately needs to create a new class object. You've thought about emitting emit class source code to a string and using a function such as `exec()` to evaluate it, but you'd prefer a more elegant solution.

### Solution

You can use the function `types.new_class()` to instantiate new class objects. All you need to do is provide the name of the class, tuple of parent classes, keyword arguments, and a callback that populates the class dictionary with members. For example:

```
# stock.py
# Example of making a class manually from parts

# Methods
def __init__(self, name, shares, price):
    self.name = name
```

```

        self.shares = shares
        self.price = price

    def cost(self):
        return self.shares * self.price

cls_dict = {
    '__init__': __init__,
    'cost': cost,
}

# Make a class
import types

Stock = types.new_class('Stock', (), {}, lambda ns: ns.update(cls_dict))
Stock.__module__ = __name__

```

This makes a normal class object that works just like you expect:

```

>>> s = Stock('ACME', 50, 91.1)
>>> s
<stock.Stock object at 0x1006a9b10>
>>> s.cost()
4555.0
>>>

```

A subtle facet of the solution is the assignment to `Stock.__module__` after the call to `types.new_class()`. Whenever a class is defined, its `__module__` attribute contains the name of the module in which it was defined. This name is used to produce the output made by methods such as `__repr__()`. It's also used by various libraries, such as `pickle`. Thus, in order for the class you make to be “proper,” you need to make sure this attribute is set accordingly.

If the class you want to create involves a different metaclass, it would be specified in the third argument to `types.new_class()`. For example:

```

>>> import abc
>>> Stock = types.new_class('Stock', (), {'metaclass': abc.ABCMeta},
...                               lambda ns: ns.update(cls_dict))
...
>>> Stock.__module__ = __name__
>>> Stock
<class '__main__.Stock'>
>>> type(Stock)
<class 'abc.ABCMeta'>
>>>

```

The third argument may also contain other keyword arguments. For example, a class definition like this

```

class Spam(Base, debug=True, typecheck=False):
    ...

```

would translate to a `new_class()` call similar to this:

```
Spam = types.new_class('Spam', (Base,),
                       {'debug': True, 'typecheck': False},
                       lambda ns: ns.update(cls_dict))
```

The fourth argument to `new_class()` is the most mysterious, but it is a function that receives the mapping object being used for the class namespace as input. This is normally a dictionary, but it's actually whatever object gets returned by the `__prepare__()` method, as described in [Recipe 9.14](#). This function should add new entries to the namespace using the `update()` method (as shown) or other mapping operations.

## Discussion

Being able to manufacture new class objects can be useful in certain contexts. One of the more familiar examples involves the `collections.namedtuple()` function. For example:

```
>>> Stock = collections.namedtuple('Stock', ['name', 'shares', 'price'])
>>> Stock
<class '__main__.Stock'>
>>>
```

`namedtuple()` uses `exec()` instead of the technique shown here. However, here is a simple variant that creates a class directly:

```
import operator
import types
import sys

def named_tuple(classname, fieldnames):
    # Populate a dictionary of field property accessors
    cls_dict = { name: property(operator.itemgetter(n))
                  for n, name in enumerate(fieldnames) }

    # Make a __new__ function and add to the class dict
    def __new__(cls, *args):
        if len(args) != len(fieldnames):
            raise TypeError('Expected {} arguments'.format(len(fieldnames)))
        return tuple.__new__(cls, args)

    cls_dict['__new__'] = __new__

    # Make the class
    cls = types.new_class(classname, (tuple,), {},
                          lambda ns: ns.update(cls_dict))

    # Set the module to that of the caller
    cls.__module__ = sys._getframe(1).f_globals['__name__']
    return cls
```

The last part of this code uses a so-called “frame hack” involving `sys._getframe()` to obtain the module name of the caller. Another example of frame hacking appears in [Recipe 2.15](#).

The following example shows how the preceding code works:

```
>>> Point = namedtuple('Point', ['x', 'y'])
>>> Point
<class '__main__.Point'>
>>> p = Point(4, 5)
>>> len(p)
2
>>> p.x
4
>>> p.y
5
>>> p.x = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>> print('%s %s' % p)
4 5
>>>
```

One important aspect of the technique used in this recipe is its proper support for metaclasses. You might be inclined to create a class directly by instantiating a metaclass directly. For example:

```
Stock = type('Stock', (), cls_dict)
```

The problem is that this approach skips certain critical steps, such as invocation of the metaclass `__prepare__()` method. By using `types.new_class()` instead, you ensure that all of the necessary initialization steps get carried out. For instance, the callback function that’s given as the fourth argument to `types.new_class()` receives the mapping object that’s returned by the `__prepare__()` method.

If you only want to carry out the preparation step, use `types.prepare_class()`. For example:

```
import types

metaclass, kwargs, ns = types.prepare_class('Stock', (), {'metaclass': type})
```

This finds the appropriate metaclass and invokes its `__prepare__()` method. The metaclass, remaining keyword arguments, and prepared namespace are then returned.

For more information, see [PEP 3115](#), as well as [the Python documentation](#).

## 9.19. Initializing Class Members at Definition Time

### Problem

You want to initialize parts of a class definition once at the time a class is defined, not when instances are created.

### Solution

Performing initialization or setup actions at the time of class definition is a classic use of metaclasses. Essentially, a metaclass is triggered at the point of a definition, at which point you can perform additional steps.

Here is an example that uses this idea to create classes similar to named tuples from the collections module:

```
import operator

class StructTupleMeta(type):
    def __init__(cls, *args, **kwargs):
        super().__init__(*args, **kwargs)
        for n, name in enumerate(cls._fields):
            setattr(cls, name, property(operator.itemgetter(n)))

class StructTuple(tuple, metaclass=StructTupleMeta):
    _fields = []
    def __new__(cls, *args):
        if len(args) != len(cls._fields):
            raise ValueError('{} arguments required'.format(len(cls._fields)))
        return super().__new__(cls, args)
```

This code allows simple tuple-based data structures to be defined, like this:

```
class Stock(StructTuple):
    _fields = ['name', 'shares', 'price']

class Point(StructTuple):
    _fields = ['x', 'y']
```

Here's how they work:

```
>>> s = Stock('ACME', 50, 91.1)
>>> s
('ACME', 50, 91.1)
>>> s[0]
'ACME'
>>> s.name
'ACME'
>>> s.shares * s.price
4555.0
>>> s.shares = 23
```



```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>
```

## Discussion

In this recipe, the `StructTupleMeta` class takes the listing of attribute names in the `_fields` class attribute and turns them into property methods that access a particular tuple slot. The operator `.itemgetter()` function creates an accessor function and the `property()` function turns it into a property.

The trickiest part of this recipe is knowing when the different initialization steps occur. The `__init__()` method in `StructTupleMeta` is only called once for each class that is defined. The `cls` argument is the class that has just been defined. Essentially, the code is using the `_fields` class variable to take the newly defined class and add some new parts to it.

The `StructTuple` class serves as a common base class for users to inherit from. The `__new__()` method in that class is responsible for making new instances. The use of `__new__()` here is a bit unusual, but is partly related to the fact that we're modifying the calling signature of tuples so that we can create instances with code that uses a normal-looking calling convention like this:

```
s = Stock('ACME', 50, 91.1)      # OK
s = Stock(('ACME', 50, 91.1))    # Error
```

Unlike `__init__()`, the `__new__()` method gets triggered before an instance is created. Since tuples are immutable, it's not possible to make any changes to them once they have been created. An `__init__()` function gets triggered too late in the instance creation process to do what we want. That's why `__new__()` has been defined.

Although this is a short recipe, careful study will reward the reader with a deep insight about how Python classes are defined, how instances are created, and the points at which different methods of metaclasses and classes are invoked.

**PEP 422** may provide an alternative means for performing the task described in this recipe. However, as of this writing, it has not been adopted or accepted. Nevertheless, it might be worth a look in case you're working with a version of Python newer than Python 3.3.

## 9.20. Implementing Multiple Dispatch with Function Annotations

### Problem

You’ve learned about function argument annotations and you have a thought that you might be able to use them to implement multiple-dispatch (method overloading) based on types. However, you’re not quite sure what’s involved (or if it’s even a good idea).

### Solution

This recipe is based on a simple observation—namely, that since Python allows arguments to be annotated, perhaps it might be possible to write code like this:

```
class Spam:
    def bar(self, x:int, y:int):
        print('Bar 1:', x, y)
    def bar(self, s:str, n:int = 0):
        print('Bar 2:', s, n)

s = Spam()
s.bar(2, 3)      # Prints Bar 1: 2 3
s.bar('hello')  # Prints Bar 2: hello 0
```

Here is the start of a solution that does just that, using a combination of metaclasses and descriptors:

```
# multiple.py

import inspect
import types

class MultiMethod:
    """
    Represents a single multimethod.
    """
    def __init__(self, name):
        self._methods = {}
        self.__name__ = name

    def register(self, meth):
        """
        Register a new method as a multimethod
        """
        sig = inspect.signature(meth)

        # Build a type signature from the method's annotations
        types = []
        for name, parm in sig.parameters.items():
            if name == 'self':
```

```

        continue
    if parm.annotation is inspect.Parameter.empty:
        raise TypeError(
            'Argument {} must be annotated with a type'.format(name)
        )
    if not isinstance(parm.annotation, type):
        raise TypeError(
            'Argument {} annotation must be a type'.format(name)
        )
    if parm.default is not inspect.Parameter.empty:
        self._methods[tuple(types)] = meth
    types.append(parm.annotation)

self._methods[tuple(types)] = meth

def __call__(self, *args):
    """
    Call a method based on type signature of the arguments
    """
    types = tuple(type(arg) for arg in args[1:])
    meth = self._methods.get(types, None)
    if meth:
        return meth(*args)
    else:
        raise TypeError('No matching method for types {}'.format(types))

def __get__(self, instance, cls):
    """
    Descriptor method needed to make calls work in a class
    """
    if instance is not None:
        return types.MethodType(self, instance)
    else:
        return self

class MultiDict(dict):
    """
    Special dictionary to build multimethods in a metaclass
    """
    def __setitem__(self, key, value):
        if key in self:
            # If key already exists, it must be a multimethod or callable
            current_value = self[key]
            if isinstance(current_value, MultiMethod):
                current_value.register(value)
            else:
                mvalue = MultiMethod(key)
                mvalue.register(current_value)
                mvalue.register(value)
                super().__setitem__(key, mvalue)
        else:
            super().__setitem__(key, value)

```

```

class MultipleMeta(type):
    '''
    Metaclass that allows multiple dispatch of methods
    '''

    def __new__(cls, clsname, bases, clsdict):
        return type.__new__(cls, clsname, bases, dict(clsdict))

    @classmethod
    def __prepare__(cls, clsname, bases):
        return MultiDict()

```

To use this class, you write code like this:

```

class Spam(metaclass=MultipleMeta):
    def bar(self, x:int, y:int):
        print('Bar 1:', x, y)
    def bar(self, s:str, n:int = 0):
        print('Bar 2:', s, n)

# Example: overloaded __init__
import time
class Date(metaclass=MultipleMeta):
    def __init__(self, year: int, month:int, day:int):
        self.year = year
        self.month = month
        self.day = day

    def __init__(self):
        t = time.localtime()
        self.__init__(t.tm_year, t.tm_mon, t.tm_mday)

```

Here is an interactive session that verifies that it works:

```

>>> s = Spam()
>>> s.bar(2, 3)
Bar 1: 2 3
>>> s.bar('hello')
Bar 2: hello 0
>>> s.bar('hello', 5)
Bar 2: hello 5
>>> s.bar(2, 'hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "multiple.py", line 42, in __call__
    raise TypeError('No matching method for types {}'.format(types))
TypeError: No matching method for types (<class 'int'>, <class 'str'>)

>>> # Overloaded __init__
>>> d = Date(2012, 12, 21)
>>> # Get today's date
>>> e = Date()
>>> e.year
2012

```

```
>>> e.month
12
>>> e.day
3
>>>
```

## Discussion

Honestly, there might be too much magic going on in this recipe to make it applicable to real-world code. However, it does dive into some of the inner workings of metaclasses and descriptors, and reinforces some of their concepts. Thus, even though you might not apply this recipe directly, some of its underlying ideas might influence other programming techniques involving metaclasses, descriptors, and function annotations.

The main idea in the implementation is relatively simple. The `MultipleMeta` metaclass uses its `__prepare__()` method to supply a custom class dictionary as an instance of `MultiDict`. Unlike a normal dictionary, `MultiDict` checks to see whether entries already exist when items are set. If so, the duplicate entries get merged together inside an instance of `MultiMethod`.

Instances of `MultiMethod` collect methods by building a mapping from type signatures to functions. During construction, function annotations are used to collect these signatures and build the mapping. This takes place in the `MultiMethod.register()` method. One critical part of this mapping is that for multimethods, types must be specified on all of the arguments or else an error occurs.

To make `MultiMethod` instances emulate a callable, the `__call__()` method is implemented. This method builds a type tuple from all of the arguments except `self`, looks up the method in the internal map, and invokes the appropriate method. The `__get__()` is required to make `MultiMethod` instances operate correctly inside class definitions. In the implementation, it's being used to create proper bound methods. For example:

```
>>> b = s.bar
>>> b
<bound method Spam.bar of <__main__.Spam object at 0x1006a46d0>>
>>> b.__self__
<__main__.Spam object at 0x1006a46d0>
>>> b.__func__
<__main__.MultiMethod object at 0x1006a4d50>
>>> b(2, 3)
Bar 1: 2 3
>>> b('hello')
Bar 2: hello 0
>>>
```

To be sure, there are a lot of moving parts to this recipe. However, it's all a little unfortunate considering how many limitations there are. For one, the solution doesn't work with keyword arguments: For example:

```

>>> s.bar(x=2, y=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __call__() got an unexpected keyword argument 'y'

>>> s.bar(s='hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __call__() got an unexpected keyword argument 's'
>>>

```

There might be some way to add such support, but it would require a completely different approach to method mapping. The problem is that the keyword arguments don't arrive in any kind of particular order. When mixed up with positional arguments, you simply get a jumbled mess of arguments that you have to somehow sort out in the `__call__()` method.

This recipe is also severely limited in its support for inheritance. For example, something like this doesn't work:

```

class A:
    pass

class B(A):
    pass

class C:
    pass

class Spam(metaclass=MultipleMeta):
    def foo(self, x:A):
        print('Foo 1:', x)

    def foo(self, x:C):
        print('Foo 2:', x)

```

The reason it fails is that the `x:A` annotation fails to match instances that are subclasses (such as instances of `B`). For example:

```

>>> s = Spam()
>>> a = A()
>>> s.foo(a)
Foo 1: <__main__.A object at 0x1006a5310>
>>> c = C()
>>> s.foo(c)
Foo 2: <__main__.C object at 0x1007a1910>
>>> b = B()
>>> s.foo(b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "multiple.py", line 44, in __call__
    raise TypeError('No matching method for types {}'.format(types))

```

```

TypeError: No matching method for types (<class '__main__.B'>,)
>>>

```

As an alternative to using metaclasses and annotations, it is possible to implement a similar recipe using decorators. For example:

```

import types

class multimethod:
    def __init__(self, func):
        self._methods = {}
        self.__name__ = func.__name__
        self._default = func

    def match(self, *types):
        def register(func):
            ndefaults = len(func.__defaults__) if func.__defaults__ else 0
            for n in range(ndefaults+1):
                self._methods[types[:len(types) - n]] = func
            return self
        return register

    def __call__(self, *args):
        types = tuple(type(arg) for arg in args[1:])
        meth = self._methods.get(types, None)
        if meth:
            return meth(*args)
        else:
            return self._default(*args)

    def __get__(self, instance, cls):
        if instance is not None:
            return types.MethodType(self, instance)
        else:
            return self

```

To use the decorator version, you would write code like this:

```

class Spam:
    @multimethod
    def bar(self, *args):
        # Default method called if no match
        raise TypeError('No matching method for bar')

    @bar.match(int, int)
    def bar(self, x, y):
        print('Bar 1:', x, y)

    @bar.match(str, int)
    def bar(self, s, n = 0):
        print('Bar 2:', s, n)

```

The decorator solution also suffers the same limitations as the previous implementation (namely, no support for keyword arguments and broken inheritance).

All things being equal, it's probably best to stay away from multiple dispatch in general-purpose code. There are special situations where it might make sense, such as in programs that are dispatching methods based on some kind of pattern matching. For example, perhaps the visitor pattern described in [Recipe 8.21](#) could be recast into a class that used multiple dispatch in some way. However, other than that, it's usually never a bad idea to stick with a more simple approach (simply use methods with different names).

Ideas concerning different ways to implement multiple dispatch have floated around the Python community for years. As a decent starting point for that discussion, see Guido van Rossum's blog post "[Five-Minute Multimethods in Python](#)".

## 9.21. Avoiding Repetitive Property Methods

### Problem

You are writing classes where you are repeatedly having to define property methods that perform common tasks, such as type checking. You would like to simplify the code so there is not so much code repetition.

### Solution

Consider a simple class where attributes are being wrapped by property methods:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        if not isinstance(value, str):
            raise TypeError('name must be a string')
        self._name = value

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, value):
```



```

if not isinstance(value, int):
    raise TypeError('age must be an int')
self._age = value

```

As you can see, a lot of code is being written simply to enforce some type assertions on attribute values. Whenever you see code like this, you should explore different ways of simplifying it. One possible approach is to make a function that simply defines the property for you and returns it. For example:

```

def typed_property(name, expected_type):
    storage_name = '_' + name

    @property
    def prop(self):
        return getattr(self, storage_name)

    @prop.setter
    def prop(self, value):
        if not isinstance(value, expected_type):
            raise TypeError('{} must be a {}'.format(name, expected_type))
        setattr(self, storage_name, value)
    return prop

# Example use
class Person:
    name = typed_property('name', str)
    age = typed_property('age', int)
    def __init__(self, name, age):
        self.name = name
        self.age = age

```

## Discussion

This recipe illustrates an important feature of inner function or closures—namely, their use in writing code that works a lot like a macro. The `typed_property()` function in this example may look a little weird, but it's really just generating the property code for you and returning the resulting property object. Thus, when it's used in a class, it operates exactly as if the code appearing inside `typed_property()` was placed into the class definition itself. Even though the property getter and setter methods are accessing local variables such as `name`, `expected_type`, and `storage_name`, that is fine—those values are held behind the scenes in a closure.

This recipe can be tweaked in an interesting manner using the `functools.partial()` function. For example, you can do this:

```

from functools import partial

String = partial(typed_property, expected_type=str)
Integer = partial(typed_property, expected_type=int)

```

```
# Example:
class Person:
    name = String('name')
    age = Integer('age')
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Here the code is starting to look a lot like some of the type system descriptor code shown in [Recipe 8.13](#).

## 9.22. Defining Context Managers the Easy Way

### Problem

You want to implement new kinds of context managers for use with the `with` statement.

### Solution

One of the most straightforward ways to write a new context manager is to use the `@contextmanager` decorator in the `contextlib` module. Here is an example of a context manager that times the execution of a code block:

```
import time
from contextlib import contextmanager

@contextmanager
def timethis(label):
    start = time.time()
    try:
        yield
    finally:
        end = time.time()
        print('{:}: {}'.format(label, end - start))

# Example use
with timethis('counting'):
    n = 10000000
    while n > 0:
        n -= 1
```

In the `timethis()` function, all of the code prior to the `yield` executes as the `__enter__()` method of a context manager. All of the code after the `yield` executes as the `__exit__()` method. If there was an exception, it is raised at the `yield` statement.

Here is a slightly more advanced context manager that implements a kind of transaction on a list object:

```
@contextmanager
def list_transaction(orig_list):
```

```

working = list(orig_list)
yield working
orig_list[:] = working

```

The idea here is that changes made to a list only take effect if an entire code block runs to completion with no exceptions. Here is an example that illustrates:

```

>>> items = [1, 2, 3]
>>> with list_transaction(items) as working:
...     working.append(4)
...     working.append(5)
...
>>> items
[1, 2, 3, 4, 5]
>>> with list_transaction(items) as working:
...     working.append(6)
...     working.append(7)
...     raise RuntimeError('oops')
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
RuntimeError: oops
>>> items
[1, 2, 3, 4, 5]
>>>

```

## Discussion

Normally, to write a context manager, you define a class with an `__enter__()` and `__exit__()` method, like this:

```

import time

class timethis:
    def __init__(self, label):
        self.label = label
    def __enter__(self):
        self.start = time.time()
    def __exit__(self, exc_ty, exc_val, exc_tb):
        end = time.time()
        print('{:}: {}'.format(self.label, end - self.start))

```

Although this isn't hard, it's a lot more tedious than writing a simple function using `@contextmanager`.

`@contextmanager` is really only used for writing self-contained context-management functions. If you have some object (e.g., a file, network connection, or lock) that needs to support the `with` statement, you still need to implement the `__enter__()` and `__exit__()` methods separately.

## 9.23. Executing Code with Local Side Effects

### Problem

You are using `exec()` to execute a fragment of code in the scope of the caller, but after execution, none of its results seem to be visible.

### Solution

To better understand the problem, try a little experiment. First, execute a fragment of code in the global namespace:

```
>>> a = 13
>>> exec('b = a + 1')
>>> print(b)
14
>>>
```

Now, try the same experiment inside a function:

```
>>> def test():
...     a = 13
...     exec('b = a + 1')
...     print(b)
...
>>> test()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in test
NameError: global name 'b' is not defined
>>>
```

As you can see, it fails with a `NameError` almost as if the `exec()` statement never actually executed. This can be a problem if you ever want to use the result of the `exec()` in a later calculation.

To fix this kind of problem, you need to use the `locals()` function to obtain a dictionary of the local variables prior to the call to `exec()`. Immediately afterward, you can extract modified values from the `locals` dictionary. For example:

```
>>> def test():
...     a = 13
...     loc = locals()
...     exec('b = a + 1')
...     b = loc['b']
...     print(b)
...
>>> test()
14
>>>
```

## Discussion

Correct use of `exec()` is actually quite tricky in practice. In fact, in most situations where you might be considering the use of `exec()`, a more elegant solution probably exists (e.g., decorators, closures, metaclasses, etc.).

However, if you still must use `exec()`, this recipe outlines some subtle aspects of using it correctly. By default, `exec()` executes code in the local and global scope of the caller. However, inside functions, the local scope passed to `exec()` is a dictionary that is a copy of the actual local variables. Thus, if the code in `exec()` makes any kind of modification, that modification is never reflected in the actual local variables. Here is another example that shows this effect:

```
>>> def test1():
...     x = 0
...     exec('x += 1')
...     print(x)
...
>>> test1()
0
>>>
```

When you call `locals()` to obtain the local variables, as shown in the solution, you get the copy of the locals that is passed to `exec()`. By inspecting the value of the dictionary after execution, you can obtain the modified values. Here is an experiment that shows this:

```
>>> def test2():
...     x = 0
...     loc = locals()
...     print('before:', loc)
...     exec('x += 1')
...     print('after:', loc)
...     print('x =', x)
...
>>> test2()
before: {'x': 0}
after: {'loc': {...}, 'x': 1}
x = 0
>>>
```

Carefully observe the output of the last step. Unless you copy the modified value from `loc` back to `x`, the variable remains unchanged.

With any use of `locals()`, you need to be careful about the order of operations. Each time it is invoked, `locals()` will take the current value of local variables and overwrite the corresponding entries in the dictionary. Observe the outcome of this experiment:

```
>>> def test3():
...     x = 0
```

```

...     loc = locals()
...     print(loc)
...     exec('x += 1')
...     print(loc)
...     locals()
...     print(loc)
...
>>> test3()
{'x': 0}
{'loc': {...}, 'x': 1}
{'loc': {...}, 'x': 0}
>>>

```

Notice how the last call to `locals()` caused `x` to be overwritten.

As an alternative to using `locals()`, you might make your own dictionary and pass it to `exec()`. For example:

```

>>> def test4():
...     a = 13
...     loc = { 'a' : a }
...     glb = { }
...     exec('b = a + 1', glb, loc)
...     b = loc['b']
...     print(b)
...
>>> test4()
14
>>>

```

For most uses of `exec()`, this is probably good practice. You just need to make sure that the global and local dictionaries are properly initialized with names that the executed code will access.

Last, but not least, before using `exec()`, you might ask yourself if other alternatives are available. Many problems where you might consider the use of `exec()` can be replaced by closures, decorators, metaclasses, or other metaprogramming features.

## 9.24. Parsing and Analyzing Python Source

### Problem

You want to write programs that parse and analyze Python source code.

### Solution

Most programmers know that Python can evaluate or execute source code provided in the form of a string. For example:

```

>>> x = 42
>>> eval('2 + 3*4 + x')
56
>>> exec('for i in range(10): print(i)')
0
1
2
3
4
5
6
7
8
9
>>>

```

However, the `ast` module can be used to compile Python source code into an abstract syntax tree (AST) that can be analyzed. For example:

```

>>> import ast
>>> ex = ast.parse('2 + 3*4 + x', mode='eval')
>>> ex
<_ast.Expression object at 0x1007473d0>
>>> ast.dump(ex)
"Expression(body=BinOp(left=BinOp(left=Num(n=2), op=Add(),
right=BinOp(left=Num(n=3), op=Mult(), right=Num(n=4))), op=Add(),
right=Name(id='x', ctx=Load())))"

>>> top = ast.parse('for i in range(10): print(i)', mode='exec')
>>> top
<_ast.Module object at 0x100747390>
>>> ast.dump(top)
"Module(body=[For(target=Name(id='i', ctx=Store()),
iter=Call(func=Name(id='range', ctx=Load()), args=[Num(n=10)],
keywords=[], starargs=None, kwargs=None),
body=[Expr(value=Call(func=Name(id='print', ctx=Load()),
args=[Name(id='i', ctx=Load())], keywords=[], starargs=None,
kwargs=None)], or_else=[])])"
>>>

```

Analyzing the source tree requires a bit of study on your part, but it consists of a collection of AST nodes. The easiest way to work with these nodes is to define a visitor class that implements various `visit_NodeName()` methods where `NodeName()` matches the node of interest. Here is an example of such a class that records information about which names are loaded, stored, and deleted.

```

import ast

class CodeAnalyzer(ast.NodeVisitor):
    def __init__(self):
        self.loaded = set()
        self.stored = set()

```

```

        self.deleted = set()
    def visit_Name(self, node):
        if isinstance(node.ctx, ast.Load):
            self.loaded.add(node.id)
        elif isinstance(node.ctx, ast.Store):
            self.stored.add(node.id)
        elif isinstance(node.ctx, ast.Del):
            self.deleted.add(node.id)

# Sample usage
if __name__ == '__main__':
    # Some Python code
    code = '''
for i in range(10):
    print(i)
del i
'''

    # Parse into an AST
    top = ast.parse(code, mode='exec')

    # Feed the AST to analyze name usage
    c = CodeAnalyzer()
    c.visit(top)
    print('Loaded:', c.loaded)
    print('Stored:', c.stored)
    print('Deleted:', c.deleted)

```

If you run this program, you'll get output like this:

```

Loaded: {'i', 'range', 'print'}
Stored: {'i'}
Deleted: {'i'}

```

Finally, ASTs can be compiled and executed using the `compile()` function. For example:

```

>>> exec(compile(top, '<stdin>', 'exec'))
0
1
2
3
4
5
6
7
8
9
>>>

```

## Discussion

The fact that you can analyze source code and get information from it could be the start of writing various code analysis, optimization, or verification tools. For instance, instead



of just blindly passing some fragment of code into a function like `exec()`, you could turn it into an AST first and look at it in some detail to see what it's doing. You could also write tools that look at the entire source code for a module and perform some sort of static analysis over it.

It should be noted that it is also possible to rewrite the AST to represent new code if you really know what you're doing. Here is an example of a decorator that lowers globally accessed names into the body of a function by reparsing the function body's source code, rewriting the AST, and recreating the function's code object:

```
# namelower.py
import ast
import inspect

# Node visitor that lowers globally accessed names into
# the function body as local variables.
class NameLower(ast.NodeVisitor):
    def __init__(self, lowered_names):
        self.lowered_names = lowered_names

    def visit_FunctionDef(self, node):
        # Compile some assignments to lower the constants
        code = '__globals = globals()\n'
        code += '\n'.join("{} = __globals['{}']".format(name)
                          for name in self.lowered_names)

        code_ast = ast.parse(code, mode='exec')

        # Inject new statements into the function body
        node.body[:0] = code_ast.body

        # Save the function object
        self.func = node

# Decorator that turns global names into locals
def lower_names(*namelist):
    def lower(func):
        srclines = inspect.getsource(func).splitlines()
        # Skip source lines prior to the @lower_names decorator
        for n, line in enumerate(srclines):
            if '@lower_names' in line:
                break

        src = '\n'.join(srclines[n+1:])
        # Hack to deal with indented code
        if src.startswith((' ', '\t')):
            src = 'if 1:\n' + src
        top = ast.parse(src, mode='exec')

        # Transform the AST
        cl = NameLower(namelist)
```

```

cl.visit(top)

# Execute the modified AST
temp = {}
exec(compile(top, '', 'exec'), temp, temp)

# Pull out the modified code object
func.__code__ = temp[func.__name__].__code__
return func
return lower

```

To use this code, you would write code such as the following:

```

INCR = 1

@lower_names('INCR')
def countdown(n):
    while n > 0:
        n -= INCR

```

The decorator rewrites the source code of the `countdown()` function to look like this:

```

def countdown(n):
    __globals = globals()
    INCR = __globals['INCR']
    while n > 0:
        n -= INCR

```

In a performance test, it makes the function run about 20% faster.

Now, should you go applying this decorator to all of your functions? Probably not. However, it's a good illustration of some very advanced things that might be possible through AST manipulation, source code manipulation, and other techniques.

This recipe was inspired by a similar recipe at [ActiveState](#) that worked by manipulating Python's byte code. Working with the AST is a higher-level approach that might be a bit more straightforward. See the next recipe for more information about byte code.

## 9.25. Disassembling Python Byte Code

### Problem

You want to know in detail what your code is doing under the covers by disassembling it into lower-level byte code used by the interpreter.

### Solution

The `dis` module can be used to output a disassembly of any Python function. For example:

```

>>> def countdown(n):
...     while n > 0:
...         print('T-minus', n)
...         n -= 1
...         print('Blastoff!')
...
>>> import dis
>>> dis.dis(countdown)
2          0 SETUP_LOOP                39 (to 42)
          >>    3 LOAD_FAST              0 (n)
              6 LOAD_CONST            1 (0)
              9 COMPARE_OP             4 (>)
             12 POP_JUMP_IF_FALSE      41

          3          15 LOAD_GLOBAL          0 (print)
              18 LOAD_CONST            2 ('T-minus')
              21 LOAD_FAST              0 (n)
              24 CALL_FUNCTION         2 (2 positional, 0 keyword pair)
              27 POP_TOP

          4          28 LOAD_FAST              0 (n)
              31 LOAD_CONST            3 (1)
              34 INPLACE_SUBTRACT
              35 STORE_FAST           0 (n)
              38 JUMP_ABSOLUTE         3
          >>    41 POP_BLOCK

          5          >>    42 LOAD_GLOBAL          0 (print)
              45 LOAD_CONST            4 ('Blastoff!')
              48 CALL_FUNCTION         1 (1 positional, 0 keyword pair)
              51 POP_TOP
              52 LOAD_CONST            0 (None)
              55 RETURN_VALUE

>>>

```

## Discussion

The `dis` module can be useful if you ever need to study what's happening in your program at a very low level (e.g., if you're trying to understand performance characteristics).

The raw byte code interpreted by the `dis()` function is available on functions as follows:

```

>>> countdown.__code__.co_code
b"x'\x00|\x00\x00d\x01\x00k\x04\x00r)\x00t\x00\x00d\x02\x00|\x00\x00\x83\x02\x00\x01|\x00\x00d\x03\x008}\x00\x00q\x03\x00wt\x00\x00d\x04\x00\x83\x01\x00\x01d\x00\x005"
>>>

```

If you ever want to interpret this code yourself, you would need to use some of the constants defined in the `opcode` module. For example:

```

>>> c = countdown.__code__.co_code
>>> import opcode
>>> opcode.opname[c[0]]
>>> opcode.opname[c[0]]
'SETUP_LOOP'
>>> opcode.opname[c[3]]
'LOAD_FAST'
>>>

```

Ironically, there is no function in the `dis` module that makes it easy for you to process the byte code in a programmatic way. However, this generator function will take the raw byte code sequence and turn it into opcodes and arguments.

```

import opcode

def generate_opcodes(codebytes):
    extended_arg = 0
    i = 0
    n = len(codebytes)
    while i < n:
        op = codebytes[i]
        i += 1
        if op >= opcode.HAVE_ARGUMENT:
            oparg = codebytes[i] + codebytes[i+1]*256 + extended_arg
            extended_arg = 0
            i += 2
            if op == opcode.EXTENDED_ARG:
                extended_arg = oparg * 65536
                continue
        else:
            oparg = None
        yield (op, oparg)

```

To use this function, you would use code like this:

```

>>> for op, oparg in generate_opcodes(countdown.__code__.co_code):
...     print(op, opcode.opname[op], oparg)
...
120 SETUP_LOOP 39
124 LOAD_FAST 0
100 LOAD_CONST 1
107 COMPARE_OP 4
114 POP_JUMP_IF_FALSE 41
116 LOAD_GLOBAL 0
100 LOAD_CONST 2
124 LOAD_FAST 0
131 CALL_FUNCTION 2
1 POP_TOP None
124 LOAD_FAST 0
100 LOAD_CONST 3
56 INPLACE_SUBTRACT None
125 STORE_FAST 0
113 JUMP_ABSOLUTE 3

```

```

87 POP_BLOCK None
116 LOAD_GLOBAL 0
100 LOAD_CONST 4
131 CALL_FUNCTION 1
1 POP_TOP None
100 LOAD_CONST 0
83 RETURN_VALUE None
>>>

```

It's a little-known fact, but you can replace the raw byte code of any function that you want. It takes a bit of work to do it, but here's an example of what's involved:

```

>>> def add(x, y):
...     return x + y
...
>>> c = add.__code__
>>> c
<code object add at 0x1007beed0, file "<stdin>", line 1>
>>> c.co_code
b'|\x00\x00|\x01\x00\x17S'
>>>
>>> # Make a completely new code object with bogus byte code
>>> import types
>>> newbytecode = b'xxxxxxx'
>>> nc = types.CodeType(c.co_argcount, c.co_kwonlyargcount,
...     c.co_nlocals, c.co_stacksize, c.co_flags, newbytecode, c.co_consts,
...     c.co_names, c.co_varnames, c.co_filename, c.co_name,
...     c.co_firstlineno, c.co_lnotab)
>>> nc
<code object add at 0x10069fe40, file "<stdin>", line 1>
>>> add.__code__ = nc
>>> add(2,3)
Segmentation fault

```

Having the interpreter crash is a pretty likely outcome of pulling a crazy stunt like this. However, developers working on advanced optimization and metaprogramming tools might be inclined to rewrite byte code for real. This last part illustrates how to do it. See [this code on ActiveState](#) for another example of such code in action.



---

# Modules and Packages

Modules and packages are the core of any large project, and the Python installation itself. This chapter focuses on common programming techniques involving modules and packages, such as how to organize packages, splitting large modules into multiple files, and creating namespace packages. Recipes that allow you to customize the operation of the `import` statement itself are also given.

## 10.1. Making a Hierarchical Package of Modules

### Problem

You want to organize your code into a package consisting of a hierarchical collection of modules.

### Solution

Making a package structure is simple. Just organize your code as you wish on the file-system and make sure that every directory defines an `__init__.py` file. For example:

```
graphics/  
    __init__.py  
    primitive/  
        __init__.py  
        line.py  
        fill.py  
        text.py  
    formats/  
        __init__.py  
        png.py  
        jpg.py
```

Once you have done this, you should be able to perform various `import` statements, such as the following:

```
import graphics.primitive.line
from graphics.primitive import line
import graphics.formats.jpg as jpg
```

## Discussion

Defining a hierarchy of modules is as easy as making a directory structure on the file-system. The purpose of the `__init__.py` files is to include optional initialization code that runs as different levels of a package are encountered. For example, if you have the statement `import graphics`, the file `graphics/__init__.py` will be imported and form the contents of the `graphics` namespace. For an import such as `import graphics.formats.jpg`, the files `graphics/__init__.py` and `graphics/formats/__init__.py` will both be imported prior to the final import of the `graphics/formats/jpg.py` file.

More often than not, it's fine to just leave the `__init__.py` files empty. However, there are certain situations where they might include code. For example, an `__init__.py` file can be used to automatically load submodules like this:

```
# graphics/formats/__init__.py

from . import jpg
from . import png
```

For such a file, a user merely has to use a single `import graphics.formats` instead of a separate import for `graphics.formats.jpg` and `graphics.formats.png`.

Other common uses of `__init__.py` include consolidating definitions from multiple files into a single logical namespace, as is sometimes done when splitting modules. This is discussed in [Recipe 10.4](#).

Astute programmers will notice that Python 3.3 still seems to perform package imports even if no `__init__.py` files are present. If you don't define `__init__.py`, you actually create what's known as a “namespace package,” which is described in [Recipe 10.5](#). All things being equal, include the `__init__.py` files if you're just starting out with the creation of a new package.

## 10.2. Controlling the Import of Everything

### Problem

You want precise control over the symbols that are exported from a module or package when a user uses the `from module import *` statement.



## Solution

Define a variable `__all__` in your module that explicitly lists the exported names. For example:

```
# somemodule.py

def spam():
    pass

def grok():
    pass

blah = 42

# Only export 'spam' and 'grok'
__all__ = ['spam', 'grok']
```

## Discussion

Although the use of `from module import *` is strongly discouraged, it still sees frequent use in modules that define a large number of names. If you don't do anything, this form of import will export all names that don't start with an underscore. On the other hand, if `__all__` is defined, then only the names explicitly listed will be exported.

If you define `__all__` as an empty list, then nothing will be exported. An `AttributeError` is raised on import if `__all__` contains undefined names.

## 10.3. Importing Package Submodules Using Relative Names

### Problem

You have code organized as a package and want to import a submodule from one of the other package submodules without hardcoding the package name into the import statement.

### Solution

To import modules of a package from other modules in the same package, use a package-relative import. For example, suppose you have a package `mypackage` organized as follows on the filesystem:

```
mypackage/
  __init__.py
  A/
    __init__.py
```

```
    spam.py
    grok.py
B/
    __init__.py
    bar.py
```

If the module `mypackage.A.spam` wants to import the module `grok` located in the same directory, it should include an import statement like this:

```
# mypackage/A/spam.py

from . import grok
```

If the same module wants to import the module `B.bar` located in a different directory, it can use an import statement like this:

```
# mypackage/A/spam.py

from ..B import bar
```

Both of the import statements shown operate relative to the location of the `spam.py` file and do not include the top-level package name.

## Discussion

Inside packages, imports involving modules in the same package can either use fully specified absolute names or a relative imports using the syntax shown. For example:

```
# mypackage/A/spam.py

from mypackage.A import grok    # OK
from . import grok              # OK
import grok                     # Error (not found)
```

The downside of using an absolute name, such as `mypackage.A`, is that it hardcodes the top-level package name into your source code. This, in turn, makes your code more brittle and hard to work with if you ever want to reorganize it. For example, if you ever changed the name of the package, you would have to go through all of your files and fix the source code. Similarly, hardcoded names make it difficult for someone else to move the code around. For example, perhaps someone wants to install two different versions of a package, differentiating them only by name. If relative imports are used, it would all work fine, whereas everything would break with absolute names.

The `.` and `..` syntax on the `import` statement might look funny, but think of it as specifying a directory name. `.` means look in the current directory and `..B` means look in the `../B` directory. This syntax only works with the `from` form of import. For example:

```
from . import grok    # OK
import .grok          # ERROR
```

Although it looks like you could navigate the filesystem using a relative import, they are not allowed to escape the directory in which a package is defined. That is, combinations of dotted name patterns that would cause an import to occur from a non-package directory cause an error.

Finally, it should be noted that relative imports only work for modules that are located inside a proper package. In particular, they do not work inside simple modules located at the top level of scripts. They also won't work if parts of a package are executed directly as a script. For example:

```
% python3 mypackage/A/spam.py      # Relative imports fail
```

On the other hand, if you execute the preceding script using the `-m` option to Python, the relative imports will work properly. For example:

```
% python3 -m mypackage.A.spam      # Relative imports work
```

For more background on relative package imports, see [PEP 328](#).

## 10.4. Splitting a Module into Multiple Files

### Problem

You have a module that you would like to split into multiple files. However, you would like to do it without breaking existing code by keeping the separate files unified as a single logical module.

### Solution

A program module can be split into separate files by turning it into a package. Consider the following simple module:

```
# mymodule.py

class A:
    def spam(self):
        print('A.spam')

class B(A):
    def bar(self):
        print('B.bar')
```

Suppose you want to split *mymodule.py* into two files, one for each class definition. To do that, start by replacing the *mymodule.py* file with a directory called *mymodule*. In that directory, create the following files:

```
mymodule/  
    __init__.py  
    a.py  
    b.py
```

In the *a.py* file, put this code:

```
# a.py  
  
class A:  
    def spam(self):  
        print('A.spam')
```

In the *b.py* file, put this code:

```
# b.py  
  
from .a import A  
  
class B(A):  
    def bar(self):  
        print('B.bar')
```

Finally, in the *\_\_init\_\_.py* file, glue the two files together:

```
# __init__.py  
  
from .a import A  
from .b import B
```

If you follow these steps, the resulting *mymodule* package will appear to be a single logical module:

```
>>> import mymodule  
>>> a = mymodule.A()  
>>> a.spam()  
A.spam  
>>> b = mymodule.B()  
>>> b.bar()  
B.bar  
>>>
```

## Discussion

The primary concern in this recipe is a design question of whether or not you want users to work with a lot of small modules or just a single module. For example, in a large code base, you could just break everything up into separate files and make users use a lot of `import` statements like this:

```
from mymodule.a import A  
from mymodule.b import B  
...
```

This works, but it places more of a burden on the user to know where the different parts are located. Often, it's just easier to unify things and allow a single import like this:

```
from mymodule import A, B
```

For this latter case, it's most common to think of `mymodule` as being one large source file. However, this recipe shows how to stitch multiple files together into a single logical namespace. The key to doing this is to create a package directory and to use the `__init__.py` file to glue the parts together.

When a module gets split, you'll need to pay careful attention to cross-filename references. For instance, in this recipe, class `B` needs to access class `A` as a base class. A package-relative import `from .a import A` is used to get it.

Package-relative imports are used throughout the recipe to avoid hardcoding the top-level module name into the source code. This makes it easier to rename the module or move it around elsewhere later (see [Recipe 10.3](#)).

One extension of this recipe involves the introduction of “lazy” imports. As shown, the `__init__.py` file imports all of the required subcomponents all at once. However, for a very large module, perhaps you only want to load components as they are needed. To do that, here is a slight variation of `__init__.py`:

```
# __init__.py

def A():
    from .a import A
    return A()

def B():
    from .b import B
    return B()
```

In this version, classes `A` and `B` have been replaced by functions that load the desired classes when they are first accessed. To a user, it won't look much different. For example:

```
>>> import mymodule
>>> a = mymodule.A()
>>> a.spam()
A.spam
>>>
```

The main downside of lazy loading is that inheritance and type checking might break. For example, you might have to change your code slightly:

```
if isinstance(x, mymodule.A):      # Error
    ...

if isinstance(x, mymodule.a.A):    # Ok
    ...
```

For a real-world example of lazy loading, look at the source code for *multiprocessing/\_\_init\_\_.py* in the standard library.

## 10.5. Making Separate Directories of Code Import Under a Common Namespace

### Problem

You have a large base of code with parts possibly maintained and distributed by different people. Each part is organized as a directory of files, like a package. However, instead of having each part installed as a separated named package, you would like all of the parts to join together under a common package prefix.

### Solution

Essentially, the problem here is that you would like to define a top-level Python package that serves as a namespace for a large collection of separately maintained subpackages. This problem often arises in large application frameworks where the framework developers want to encourage users to distribute plug-ins or add-on packages.

To unify separate directories under a common namespace, you organize the code just like a normal Python package, but you omit *\_\_init\_\_.py* files in the directories where the components are going to join together. To illustrate, suppose you have two different directories of Python code like this:

```
foo-package/  
  spam/  
    blah.py  
  
bar-package/  
  spam/  
    grok.py
```

In these directories, the name *spam* is being used as a common namespace. Observe that there is no *\_\_init\_\_.py* file in either directory.

Now watch what happens if you add both *foo-package* and *bar-package* to the Python module path and try some imports:

```
>>> import sys  
>>> sys.path.extend(['foo-package', 'bar-package'])  
>>> import spam.blah  
>>> import spam.grok  
>>>
```

You'll observe that, by magic, the two different package directories merge together and you can import either *spam.blah* or *spam.grok*. It just works.

## Discussion

The mechanism at work here is a feature known as a “namespace package.” Essentially, a namespace package is a special kind of package designed for merging different directories of code together under a common namespace, as shown. For large frameworks, this can be useful, since it allows parts of a framework to be broken up into separately installed downloads. It also enables people to easily make third-party add-ons and other extensions to such frameworks.

The key to making a namespace package is to make sure there are no `__init__.py` files in the top-level directory that is to serve as the common namespace. The missing `__init__.py` file causes an interesting thing to happen on package import. Instead of causing an error, the interpreter instead starts creating a list of all directories that happen to contain a matching package name. A special namespace package module is then created and a read-only copy of the list of directories is stored in its `__path__` variable. For example:

```
>>> import spam
>>> spam.__path__
NamespacePath(['foo-package/spam', 'bar-package/spam'])
>>>
```

The directories on `__path__` are used when locating further package subcomponents (e.g., when importing `spam.grok` or `spam.blah`).

An important feature of namespace packages is that anyone can extend the namespace with their own code. For example, suppose you made your own directory of code like this:

```
my-package/
  spam/
    custom.py
```

If you added your directory of code to `sys.path` along with the other packages, it would just seamlessly merge together with the other `spam` package directories:

```
>>> import spam.custom
>>> import spam.grok
>>> import spam.blah
>>>
```

As a debugging tool, the main way that you can tell if a package is serving as a namespace package is to check its `__file__` attribute. If it’s missing altogether, the package is a namespace. This will also be indicated in the representation string by the word “namespace”:

```
>>> spam.__file__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute '__file__'
```

```
>>> spam
<module 'spam' (namespace)>
>>>
```

Further information about namespace packages can be found in [PEP 420](#).

## 10.6. Reloading Modules

### Problem

You want to reload an already loaded module because you’ve made changes to its source.

### Solution

To reload a previously loaded module, use `imp.reload()`. For example:

```
>>> import spam
>>> import imp
>>> imp.reload(spam)
<module 'spam' from './spam.py'>
>>>
```

### Discussion

Reloading a module is something that is often useful during debugging and development, but which is generally never safe in production code due to the fact that it doesn’t always work as you expect.

Under the covers, the `reload()` operation wipes out the contents of a module’s underlying dictionary and refreshes it by re-executing the module’s source code. The identity of the module object itself remains unchanged. Thus, this operation updates the module everywhere that it has been imported in a program.

However, `reload()` does not update definitions that have been imported using statements such as `from module import name`. To illustrate, consider the following code:

```
# spam.py

def bar():
    print('bar')

def grok():
    print('grok')
```

Now start an interactive session:

```
>>> import spam
>>> from spam import grok
>>> spam.bar()
bar
```



```
>>> grok()
grok
>>>
```

Without quitting Python, go edit the source code to *spam.py* so that the function `grok()` looks like this:

```
def grok():
    print('New grok')
```

Now go back to the interactive session, perform a reload, and try this experiment:

```
>>> import imp
>>> imp.reload(spam)
<module 'spam' from './spam.py'>
>>> spam.bar()
bar
>>> grok()                # Notice old output
grok
>>> spam.grok()           # Notice new output
New grok
>>>
```

In this example, you'll observe that there are two versions of the `grok()` function loaded. Generally, this is not what you want, and is just the sort of thing that eventually leads to massive headaches.

For this reason, reloading of modules is probably something to be avoided in production code. Save it for debugging or for interactive sessions where you're experimenting with the interpreter and trying things out.

## 10.7. Making a Directory or Zip File Runnable As a Main Script

### Problem

You have a program that has grown beyond a simple script into an application involving multiple files. You'd like to have some easy way for users to run the program.

### Solution

If your application program has grown into multiple files, you can put it into its own directory and add a `__main__.py` file. For example, you can create a directory like this:

```
myapplication/
  spam.py
  bar.py
  grok.py
  __main__.py
```

If `__main__.py` is present, you can simply run the Python interpreter on the top-level directory like this:

```
bash % python3 myapplication
```

The interpreter will execute the `__main__.py` file as the main program.

This technique also works if you package all of your code up into a zip file. For example:

```
bash % ls
spam.py  bar.py  grok.py  __main__.py
bash % zip -r myapp.zip *.py
bash % python3 myapp.zip
... output from __main__.py ...
```

## Discussion

Creating a directory or zip file and adding a `__main__.py` file is one possible way to package a larger Python application. It's a little bit different than a package in that the code isn't meant to be used as a standard library module that's installed into the Python library. Instead, it's just this bundle of code that you want to hand someone to execute.

Since directories and zip files are a little different than normal files, you may also want to add a supporting shell script to make execution easier. For example, if the code was in a file named `myapp.zip`, you could make a top-level script like this:

```
#!/usr/bin/env python3 /usr/local/bin/myapp.zip
```

# 10.8. Reading Datafiles Within a Package

## Problem

Your package includes a datafile that your code needs to read. You need to do this in the most portable way possible.

## Solution

Suppose you have a package with files organized as follows:

```
mypackage/
__init__.py
somedata.dat
spam.py
```

Now suppose the file `spam.py` wants to read the contents of the file `somedata.dat`. To do it, use the following code:

```
# spam.py

import pkgutil
data = pkgutil.get_data(__package__, 'somedata.dat')
```

The resulting variable `data` will be a byte string containing the raw contents of the file.

## Discussion

To read a datafile, you might be inclined to write code that uses built-in I/O functions, such as `open()`. However, there are several problems with this approach.

First, a package has very little control over the current working directory of the interpreter. Thus, any I/O operations would have to be programmed to use absolute filenames. Since each module includes a `__file__` variable with the full path, it's not impossible to figure out the location, but it's messy.

Second, packages are often installed as *.zip* or *.egg* files, which don't preserve the files in the same way as a normal directory on the filesystem. Thus, if you tried to use `open()` on a datafile contained in an archive, it wouldn't work at all.

The `pkgutil.get_data()` function is meant to be a high-level tool for getting a datafile regardless of where or how a package has been installed. It will simply “work” and return the file contents back to you as a byte string.

The first argument to `get_data()` is a string containing the package name. You can either supply it directly or use a special variable, such as `__package__`. The second argument is the relative name of the file within the package. If necessary, you can navigate into different directories using standard Unix filename conventions as long as the final directory is still located within the package.

## 10.9. Adding Directories to `sys.path`

### Problem

You have Python code that can't be imported because it's not located in a directory listed in `sys.path`. You would like to add new directories to Python's path, but don't want to hardwire it into your code.

### Solution

There are two common ways to get new directories added to `sys.path`. First, you can add them through the use of the `PYTHONPATH` environment variable. For example:

```
bash % env PYTHONPATH=/some/dir:/other/dir python3
Python 3.3.0 (default, Oct  4 2012, 10:17:33)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
```

```
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.path
['', '/some/dir', '/other/dir', ...]
>>>
```

In a custom application, this environment variable could be set at program startup or through a shell script of some kind.

The second approach is to create a *.pth* file that lists the directories like this:

```
# myapplication.pth
/some/dir
/other/dir
```

This *.pth* file needs to be placed into one of Python's *site-packages* directories, which are typically located at */usr/local/lib/python3.3/site-packages* or *~/.local/lib/python3.3/site-packages*. On interpreter startup, the directories listed in the *.pth* file will be added to *sys.path* as long as they exist on the filesystem. Installation of a *.pth* file might require administrator access if it's being added to the system-wide Python interpreter.

## Discussion

Faced with trouble locating files, you might be inclined to write code that manually adjusts the value of *sys.path*. For example:

```
import sys
sys.path.insert(0, '/some/dir')
sys.path.insert(0, '/other/dir')
```

Although this “works,” it is extremely fragile in practice and should be avoided if possible. Part of the problem with this approach is that it adds hardcoded directory names to your source. This can cause maintenance problems if your code ever gets moved around to a new location. It's usually much better to configure the path elsewhere in a manner that can be adjusted without making source code edits.

You can sometimes work around the problem of hardcoded directories if you carefully construct an appropriate absolute path using module-level variables, such as *\_\_file\_\_*. For example:

```
import sys
from os.path import abspath, join, dirname
sys.path.insert(0, abspath(dirname('__file__'), 'src'))
```

This adds an *src* directory to the path where that directory is located in the same directory as the code that's executing the insertion step.

The *site-packages* directories are the locations where third-party modules and packages normally get installed. If your code was installed in that manner, that's where it would be placed. Although *.pth* files for configuring the path must appear in *site-packages*, they

can refer to any directories on the system that you wish. Thus, you can elect to have your code in a completely different set of directories as long as those directories are included in a *.pth* file.

## 10.10. Importing Modules Using a Name Given in a String

### Problem

You have the name of a module that you would like to import, but it's being held in a string. You would like to invoke the `import` command on the string.

### Solution

Use the `importlib.import_module()` function to manually import a module or part of a package where the name is given as a string. For example:

```
>>> import importlib
>>> math = importlib.import_module('math')
>>> math.sin(2)
0.9092974268256817
>>> mod = importlib.import_module('urllib.request')
>>> u = mod.urlopen('http://www.python.org')
>>>
```

`import_module` simply performs the same steps as `import`, but returns the resulting module object back to you as a result. You just need to store it in a variable and use it like a normal module afterward.

If you are working with packages, `import_module()` can also be used to perform relative imports. However, you need to give it an extra argument. For example:

```
import importlib

# Same as 'from . import b'
b = importlib.import_module('.b', __package__)
```

### Discussion

The problem of manually importing modules with `import_module()` most commonly arises when writing code that manipulates or wraps around modules in some way. For example, perhaps you're implementing a customized importing mechanism of some kind where you need to load a module by name and perform patches to the loaded code.

In older code, you will sometimes see the built-in `__import__()` function used to perform imports. Although this works, `importlib.import_module()` is usually easier to use.

See [Recipe 10.11](#) for an advanced example of customizing the import process.

## 10.11. Loading Modules from a Remote Machine Using Import Hooks

### Problem

You would like to customize Python's import statement so that it can transparently load modules from a remote machine.

### Solution

First, a serious disclaimer about security. The idea discussed in this recipe would be wholly bad without some kind of extra security and authentication layer. That said, the main goal is actually to take a deep dive into the inner workings of Python's `import` statement. If you get this recipe to work and understand the inner workings, you'll have a solid foundation of customizing `import` for almost any other purpose. With that out of the way, let's carry on.

At the core of this recipe is a desire to extend the functionality of the `import` statement. There are several approaches for doing this, but for the purposes of illustration, start by making the following directory of Python code:

```
testcode/  
  spam.py  
  fib.py  
  grok/  
    __init__.py  
    blah.py
```

The content of these files doesn't matter, but put a few simple statements and functions in each file so you can test them and see output when they're imported. For example:

```
# spam.py  
print("I'm spam")  
  
def hello(name):  
    print('Hello %s' % name)  
  
# fib.py  
print("I'm fib")  
  
def fib(n):  
    if n < 2:  
        return 1  
    else:  
        return fib(n-1) + fib(n-2)  
  
# grok/__init__.py  
print("I'm grok.__init__")
```

```
# grok/blah.py
print("I'm grok.blah")
```

The goal here is to allow remote access to these files as modules. Perhaps the easiest way to do this is to publish them on a web server. Simply go to the *testcode* directory and run Python like this:

```
bash % cd testcode
bash % python3 -m http.server 15000
Serving HTTP on 0.0.0.0 port 15000 ...
```

Leave that server running and start up a separate Python interpreter. Make sure you can access the remote files using `urllib`. For example:

```
>>> from urllib.request import urlopen
>>> u = urlopen('http://localhost:15000/fib.py')
>>> data = u.read().decode('utf-8')
>>> print(data)
# fib.py
print("I'm fib")

def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)

>>>
```

Loading source code from this server is going to form the basis for the remainder of this recipe. Specifically, instead of manually grabbing a file of source code using `urlopen()`, the `import` statement will be customized to do it transparently behind the scenes.

The first approach to loading a remote module is to create an explicit loading function for doing it. For example:

```
import imp
import urllib.request
import sys

def load_module(url):
    u = urllib.request.urlopen(url)
    source = u.read().decode('utf-8')
    mod = sys.modules.setdefault(url, imp.new_module(url))
    code = compile(source, url, 'exec')
    mod.__file__ = url
    mod.__package__ = ''
    exec(code, mod.__dict__)
    return mod
```

This function merely downloads the source code, compiles it into a code object using `compile()`, and executes it in the dictionary of a newly created module object. Here's how you would use the function:

```
>>> fib = load_module('http://localhost:15000/fib.py')
I'm fib
>>> fib.fib(10)
89
>>> spam = load_module('http://localhost:15000/spam.py')
I'm spam
>>> spam.hello('Guido')
Hello Guido
>>> fib
<module 'http://localhost:15000/fib.py' from 'http://localhost:15000/fib.py'>
>>> spam
<module 'http://localhost:15000/spam.py' from 'http://localhost:15000/spam.py'>
>>>
```

As you can see, it “works” for simple modules. However, it's not plugged into the usual `import` statement, and extending the code to support more advanced constructs, such as packages, would require additional work.

A much slicker approach is to create a custom importer. The first way to do this is to create what's known as a meta path importer. Here is an example:

```
# urlimport.py

import sys
import importlib.abc
import imp
from urllib.request import urlopen
from urllib.error import HTTPError, URLError
from html.parser import HTMLParser

# Debugging
import logging
log = logging.getLogger(__name__)

# Get links from a given URL
def _get_links(url):
    class LinkParser(HTMLParser):
        def handle_starttag(self, tag, attrs):
            if tag == 'a':
                attrs = dict(attrs)
                links.add(attrs.get('href').rstrip('/'))

    links = set()
    try:
        log.debug('Getting links from %s' % url)
        u = urlopen(url)
        parser = LinkParser()
        parser.feed(u.read().decode('utf-8'))
```



```

except Exception as e:
    log.debug('Could not get links. %s', e)
log.debug('links: %r', links)
return links

class UrlMetaFinder(importlib.abc.MetaPathFinder):
    def __init__(self, baseurl):
        self._baseurl = baseurl
        self._links = { }
        self._loaders = { baseurl : UrlModuleLoader(baseurl) }

    def find_module(self, fullname, path=None):
        log.debug('find_module: fullname=%r, path=%r', fullname, path)
        if path is None:
            baseurl = self._baseurl
        else:
            if not path[0].startswith(self._baseurl):
                return None
            baseurl = path[0]

        parts = fullname.split('.')
        basename = parts[-1]
        log.debug('find_module: baseurl=%r, basename=%r', baseurl, basename)

        # Check link cache
        if basename not in self._links:
            self._links[baseurl] = _get_links(baseurl)

        # Check if it's a package
        if basename in self._links[baseurl]:
            log.debug('find_module: trying package %r', fullname)
            fullurl = self._baseurl + '/' + basename
            # Attempt to load the package (which accesses __init__.py)
            loader = UrlPackageLoader(fullurl)
            try:
                loader.load_module(fullname)
                self._links[fullurl] = _get_links(fullurl)
                self._loaders[fullurl] = UrlModuleLoader(fullurl)
                log.debug('find_module: package %r loaded', fullname)
            except ImportError as e:
                log.debug('find_module: package failed. %s', e)
                loader = None
            return loader

        # A normal module
        filename = basename + '.py'
        if filename in self._links[baseurl]:
            log.debug('find_module: module %r found', fullname)
            return self._loaders[baseurl]
        else:
            log.debug('find_module: module %r not found', fullname)
            return None

```

```

def invalidate_caches(self):
    log.debug('invalidating link cache')
    self._links.clear()

# Module Loader for a URL
class UrlModuleLoader(importlib.abc.SourceLoader):
    def __init__(self, baseurl):
        self._baseurl = baseurl
        self._source_cache = {}

    def module_repr(self, module):
        return '<urlmodule %r from %r>' % (module.__name__, module.__file__)

# Required method
def load_module(self, fullname):
    code = self.get_code(fullname)
    mod = sys.modules.setdefault(fullname, imp.new_module(fullname))
    mod.__file__ = self.get_filename(fullname)
    mod.__loader__ = self
    mod.__package__ = fullname.rpartition('.')[0]
    exec(code, mod.__dict__)
    return mod

# Optional extensions
def get_code(self, fullname):
    src = self.get_source(fullname)
    return compile(src, self.get_filename(fullname), 'exec')

def get_data(self, path):
    pass

def get_filename(self, fullname):
    return self._baseurl + '/' + fullname.split('.')[0] + '.py'

def get_source(self, fullname):
    filename = self.get_filename(fullname)
    log.debug('loader: reading %r', filename)
    if filename in self._source_cache:
        log.debug('loader: cached %r', filename)
        return self._source_cache[filename]
    try:
        u = urlopen(filename)
        source = u.read().decode('utf-8')
        log.debug('loader: %r loaded', filename)
        self._source_cache[filename] = source
        return source
    except (HTTPError, URLError) as e:
        log.debug('loader: %r failed. %s', filename, e)
        raise ImportError("Can't load %s" % filename)

```

```

def is_package(self, fullname):
    return False

# Package loader for a URL
class UrlPackageLoader(UrlModuleLoader):
    def load_module(self, fullname):
        mod = super().load_module(fullname)
        mod.__path__ = [ self._baseurl ]
        mod.__package__ = fullname

    def get_filename(self, fullname):
        return self._baseurl + '/' + '__init__.py'

    def is_package(self, fullname):
        return True

# Utility functions for installing/uninstalling the loader
_installed_meta_cache = { }
def install_meta(address):
    if address not in _installed_meta_cache:
        finder = UrlMetaFinder(address)
        _installed_meta_cache[address] = finder
        sys.meta_path.append(finder)
        log.debug('%r installed on sys.meta_path', finder)

def remove_meta(address):
    if address in _installed_meta_cache:
        finder = _installed_meta_cache.pop(address)
        sys.meta_path.remove(finder)
        log.debug('%r removed from sys.meta_path', finder)

```

Here is an interactive session showing how to use the preceding code:

```

>>> # importing currently fails
>>> import fib
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'

>>> # Load the importer and retry (it works)
>>> import urlimport
>>> urlimport.install_meta('http://localhost:15000')
>>> import fib
I'm fib
>>> import spam
I'm spam
>>> import grok.blah
I'm grok.__init__
I'm grok.blah
>>> grok.blah.__file__
'http://localhost:15000/grok/blah.py'
>>>

```

This particular solution involves installing an instance of a special finder object `UrlMetaFinder` as the last entry in `sys.meta_path`. Whenever modules are imported, the finders in `sys.meta_path` are consulted in order to locate the module. In this example, the `UrlMetaFinder` instance becomes a finder of last resort that's triggered when a module can't be found in any of the normal locations.

As for the general implementation approach, the `UrlMetaFinder` class wraps around a user-specified URL. Internally, the finder builds sets of valid links by scraping them from the given URL. When imports are made, the module name is compared against this set of known links. If a match can be found, a separate `UrlModuleLoader` class is used to load source code from the remote machine and create the resulting module object. One reason for caching the links is to avoid unnecessary HTTP requests on repeated imports.

The second approach to customizing import is to write a hook that plugs directly into the `sys.path` variable, recognizing certain directory naming patterns. Add the following class and support functions to *urlimport.py*:

```
# urlimport.py

# ... include previous code above ...

# Path finder class for a URL
class UrlPathFinder(importlib.abc.PathEntryFinder):
    def __init__(self, baseurl):
        self._links = None
        self._loader = UrlModuleLoader(baseurl)
        self._baseurl = baseurl

    def find_loader(self, fullname):
        log.debug('find_loader: %r', fullname)
        parts = fullname.split('.')
        basename = parts[-1]
        # Check link cache
        if self._links is None:
            self._links = [] # See discussion
            self._links = _get_links(self._baseurl)

        # Check if it's a package
        if basename in self._links:
            log.debug('find_loader: trying package %r', fullname)
            fullurl = self._baseurl + '/' + basename
            # Attempt to load the package (which accesses __init__.py)
            loader = UrlPackageLoader(fullurl)
            try:
                loader.load_module(fullname)
                log.debug('find_loader: package %r loaded', fullname)
            except ImportError as e:
                log.debug('find_loader: %r is a namespace package', fullname)
```

```

        loader = None
        return (loader, [fullurl])

    # A normal module
    filename = basename + '.py'
    if filename in self._links:
        log.debug('find_loader: module %r found', fullname)
        return (self._loader, [])
    else:
        log.debug('find_loader: module %r not found', fullname)
        return (None, [])

    def invalidate_caches(self):
        log.debug('invalidating link cache')
        self._links = None

    # Check path to see if it looks like a URL
    _url_path_cache = {}
    def handle_url(path):
        if path.startswith(('http://', 'https://')):
            log.debug('Handle path? %s. [Yes]', path)
            if path in _url_path_cache:
                finder = _url_path_cache[path]
            else:
                finder = UrlPathFinder(path)
                _url_path_cache[path] = finder
            return finder
        else:
            log.debug('Handle path? %s. [No]', path)

    def install_path_hook():
        sys.path_hooks.append(handle_url)
        sys.path_importer_cache.clear()
        log.debug('Installing handle_url')

    def remove_path_hook():
        sys.path_hooks.remove(handle_url)
        sys.path_importer_cache.clear()
        log.debug('Removing handle_url')

```

To use this path-based finder, you simply add URLs to `sys.path`. For example:

```

>>> # Initial import fails
>>> import fib
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'

>>> # Install the path hook
>>> import urlimport
>>> urlimport.install_path_hook()

>>> # Imports still fail (not on path)

```

```

>>> import fib
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'

>>> # Add an entry to sys.path and watch it work
>>> import sys
>>> sys.path.append('http://localhost:15000')
>>> import fib
I'm fib
>>> import grok.blah
I'm grok.__init__
I'm grok.blah
>>> grok.blah.__file__
'http://localhost:15000/grok/blah.py'
>>>

```

The key to this last example is the `handle_url()` function, which is added to the `sys.path_hooks` variable. When the entries on `sys.path` are being processed, the functions in `sys.path_hooks` are invoked. If any of those functions return a finder object, that finder is used to try to load modules for that entry on `sys.path`.

It should be noted that the remotely imported modules work exactly like any other module. For instance:

```

>>> fib
<urlmodule 'fib' from 'http://localhost:15000/fib.py'>
>>> fib.__name__
'fib'
>>> fib.__file__
'http://localhost:15000/fib.py'
>>> import inspect
>>> print(inspect.getsource(fib))
# fib.py
print("I'm fib")

def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)

>>>

```

## Discussion

Before discussing this recipe in further detail, it should be emphasized that Python's module, package, and import mechanism is one of the most complicated parts of the entire language—often poorly understood by even the most seasoned Python programmers unless they've devoted effort to peeling back the covers. There are several

critical documents that are worth reading, including the documentation for the `importlib` module and [PEP 302](#). That documentation won't be repeated here, but some essential highlights will be discussed.

First, if you want to create a new module object, you use the `imp.new_module()` function. For example:

```
>>> import imp
>>> m = imp.new_module('spam')
>>> m
<module 'spam'>
>>> m.__name__
'spam'
>>>
```

Module objects usually have a few expected attributes, including `__file__` (the name of the file that the module was loaded from) and `__package__` (the name of the enclosing package, if any).

Second, modules are cached by the interpreter. The module cache can be found in the dictionary `sys.modules`. Because of this caching, it's common to combine caching and module creation together into a single step. For example:

```
>>> import sys
>>> import imp
>>> m = sys.modules.setdefault('spam', imp.new_module('spam'))
>>> m
<module 'spam'>
>>>
```

The main reason for doing this is that if a module with the given name already exists, you'll get the already created module instead. For example:

```
>>> import math
>>> m = sys.modules.setdefault('math', imp.new_module('math'))
>>> m
<module 'math' from '/usr/local/lib/python3.3/lib-dynload/math.so'>
>>> m.sin(2)
0.9092974268256817
>>> m.cos(2)
-0.4161468365471424
>>>
```

Since creating modules is easy, it is straightforward to write simple functions, such as the `load_module()` function in the first part of this recipe. A downside of this approach is that it is actually rather tricky to handle more complicated cases, such as package imports. In order to handle a package, you would have to reimplement much of the underlying logic that's already part of the normal `import` statement (e.g., checking for directories, looking for `__init__.py` files, executing those files, setting up paths, etc.).

This complexity is one of the reasons why it's often better to extend the `import` statement directly rather than defining a custom function.

Extending the `import` statement is straightforward, but involves a number of moving parts. At the highest level, `import` operations are processed by a list of “meta-path” finders that you can find in the list `sys.meta_path`. If you output its value, you'll see the following:

```
>>> from pprint import pprint
>>> pprint(sys.meta_path)
[<class '_frozen_importlib.BuiltinImporter'>,
 <class '_frozen_importlib.FrozenImporter'>,
 <class '_frozen_importlib.PathFinder'>]
>>>
```

When executing a statement such as `import fib`, the interpreter walks through the finder objects on `sys.meta_path` and invokes their `find_module()` method in order to locate an appropriate module loader. It helps to see this by experimentation, so define the following class and try the following:

```
>>> class Finder:
...     def find_module(self, fullname, path):
...         print('Looking for', fullname, path)
...         return None
...
>>> import sys
>>> sys.meta_path.insert(0, Finder())    # Insert as first entry
>>> import math
Looking for math None
>>> import types
Looking for types None
>>> import threading
Looking for threading None
Looking for time None
Looking for traceback None
Looking for linecache None
Looking for tokenize None
Looking for token None
>>>
```

Notice how the `find_module()` method is being triggered on every `import`. The role of the `path` argument in this method is to handle packages. When packages are imported, it is a list of the directories that are found in the package's `__path__` attribute. These are the paths that need to be checked to find package subcomponents. For example, notice the path setting for `xml.etree` and `xml.etree.ElementTree`:

```
>>> import xml.etree.ElementTree
Looking for xml None
Looking for xml.etree ['/usr/local/lib/python3.3/xml']
Looking for xml.etree.ElementTree ['/usr/local/lib/python3.3/xml/etree']
Looking for warnings None
```



```

Looking for contextlib None
Looking for xml.etree.ElementPath ['/usr/local/lib/python3.3/xml/etree']
Looking for _elementtree None
Looking for copy None
Looking for org None
Looking for pyexpat None
Looking for ElementC14N None
>>>

```

The placement of the finder on `sys.meta_path` is critical. Remove it from the front of the list to the end of the list and try more imports:

```

>>> del sys.meta_path[0]
>>> sys.meta_path.append(Finder())
>>> import urllib.request
>>> import datetime

```

Now you don't see any output because the imports are being handled by other entries in `sys.meta_path`. In this case, you would only see it trigger when nonexistent modules are imported:

```

>>> import fib
Looking for fib None
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'
>>> import xml.superfast
Looking for xml.superfast ['/usr/local/lib/python3.3/xml']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'xml.superfast'
>>>

```

The fact that you can install a finder to catch unknown modules is the key to the `UrlMetaFinder` class in this recipe. An instance of `UrlMetaFinder` is added to the end of `sys.meta_path`, where it serves as a kind of importer of last resort. If the requested module name can't be located by any of the other import mechanisms, it gets handled by this finder. Some care needs to be taken when handling packages. Specifically, the value presented in the path argument needs to be checked to see if it starts with the URL registered in the finder. If not, the submodule must belong to some other finder and should be ignored.

Additional handling of packages is found in the `UrlPackageLoader` class. This class, rather than importing the package name, tries to load the underlying `__init__.py` file. It also sets the module `__path__` attribute. This last part is critical, as the value set will be passed to subsequent `find_module()` calls when loading package submodules.

The path-based import hook is an extension of these ideas, but based on a somewhat different mechanism. As you know, `sys.path` is a list of directories where Python looks for modules. For example:

```

>>> from pprint import pprint
>>> import sys
>>> pprint(sys.path)
['',
 '/usr/local/lib/python3.3.zip',
 '/usr/local/lib/python3.3',
 '/usr/local/lib/python3.3/plat-darwin',
 '/usr/local/lib/python3.3/lib-dynload',
 '/usr/local/lib/...3.3/site-packages']
>>>

```

Each entry in `sys.path` is additionally attached to a finder object. You can view these finders by looking at `sys.path_importer_cache`:

```

>>> pprint(sys.path_importer_cache)
{'.': FileFinder('.'),
 '/usr/local/lib/python3.3': FileFinder('/usr/local/lib/python3.3'),
 '/usr/local/lib/python3.3/': FileFinder('/usr/local/lib/python3.3/'),
 '/usr/local/lib/python3.3/collections': FileFinder('...python3.3/collections'),
 '/usr/local/lib/python3.3/encodings': FileFinder('...python3.3/encodings'),
 '/usr/local/lib/python3.3/lib-dynload': FileFinder('...python3.3/lib-dynload'),
 '/usr/local/lib/python3.3/plat-darwin': FileFinder('...python3.3/plat-darwin'),
 '/usr/local/lib/python3.3/site-packages': FileFinder('...python3.3/site-packages'),
 '/usr/local/lib/python3.3.zip': None}
>>>

```

`sys.path_importer_cache` tends to be much larger than `sys.path` because it records finders for all known directories where code is being loaded. This includes subdirectories of packages which usually aren't included on `sys.path`.

To execute `import fib`, the directories on `sys.path` are checked in order. For each directory, the name `fib` is presented to the associated finder found in `sys.path_importer_cache`. This is also something that you can investigate by making your own finder and putting an entry in the cache. Try this experiment:

```

>>> class Finder:
...     def find_loader(self, name):
...         print('Looking for', name)
...         return (None, [])
...
>>> import sys
>>> # Add a "debug" entry to the importer cache
>>> sys.path_importer_cache['debug'] = Finder()
>>> # Add a "debug" directory to sys.path
>>> sys.path.insert(0, 'debug')
>>> import threading
Looking for threading
Looking for time
Looking for traceback
Looking for linecache
Looking for tokenize

```

```
Looking for token
>>>
```

Here, you've installed a new cache entry for the name `debug` and installed the name `debug` as the first entry on `sys.path`. On all subsequent imports, you see your finder being triggered. However, since it returns `(None, [])`, processing simply continues to the next entry.

The population of `sys.path_importer_cache` is controlled by a list of functions stored in `sys.path_hooks`. Try this experiment, which clears the cache and adds a new path checking function to `sys.path_hooks`:

```
>>> sys.path_importer_cache.clear()
>>> def check_path(path):
...     print('Checking', path)
...     raise ImportError()
...
>>> sys.path_hooks.insert(0, check_path)
>>> import fib
Checked debug
Checking .
Checking /usr/local/lib/python3.3.zip
Checking /usr/local/lib/python3.3
Checking /usr/local/lib/python3.3/plat-darwin
Checking /usr/local/lib/python3.3/lib-dynload
Checking /Users/beazley/.local/lib/python3.3/site-packages
Checking /usr/local/lib/python3.3/site-packages
Looking for fib
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'
>>>
```

As you can see, the `check_path()` function is being invoked for every entry on `sys.path`. However, since an `ImportError` exception is raised, nothing else happens (checking just moves to the next function on `sys.path_hooks`).

Using this knowledge of how `sys.path` is processed, you can install a custom path checking function that looks for filename patterns, such as URLs. For instance:

```
>>> def check_url(path):
...     if path.startswith('http://'):
...         return Finder()
...     else:
...         raise ImportError()
...
>>> sys.path.append('http://localhost:15000')
>>> sys.path_hooks[0] = check_url
>>> import fib
Looking for fib          # Finder output!
Traceback (most recent call last):
```

```

File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'

>>> # Notice installation of Finder in sys.path_importer_cache
>>> sys.path_importer_cache['http://localhost:15000']
<__main__.Finder object at 0x10064c850>
>>>

```

This is the key mechanism at work in the last part of this recipe. Essentially, a custom path checking function has been installed that looks for URLs in `sys.path`. When they are encountered, a new `UrlPathFinder` instance is created and installed into `sys.path_importer_cache`. From that point forward, all import statements that pass through that part of `sys.path` will try to use your custom finder.

Package handling with a path-based importer is somewhat tricky, and relates to the return value of the `find_loader()` method. For simple modules, `find_loader()` returns a tuple (`loader`, `None`) where `loader` is an instance of a loader that will import the module.

For a normal package, `find_loader()` returns a tuple (`loader`, `path`) where `loader` is the loader instance that will import the package (and execute `__init__.py`) and `path` is a list of the directories that will make up the initial setting of the package's `__path__` attribute. For example, if the base URL was `http://localhost:15000` and a user executed `import grok`, the path returned by `find_loader()` would be [ `'http://localhost:15000/grok'` ].

The `find_loader()` must additionally account for the possibility of a namespace package. A namespace package is a package where a valid package directory name exists, but no `__init__.py` file can be found. For this case, `find_loader()` must return a tuple (`None`, `path`) where `path` is a list of directories that would have made up the package's `__path__` attribute had it defined an `__init__.py` file. For this case, the import mechanism moves on to check further directories on `sys.path`. If more namespace packages are found, all of the resulting paths are joined together to make a final namespace package. See [Recipe 10.5](#) for more information on namespace packages.

There is a recursive element to package handling that is not immediately obvious in the solution, but also at work. All packages contain an internal path setting, which can be found in `__path__` attribute. For example:

```

>>> import xml.etree.ElementTree
>>> xml.__path__
['/usr/local/lib/python3.3/xml']
>>> xml.etree.__path__
['/usr/local/lib/python3.3/xml/etree']
>>>

```

As mentioned, the setting of `__path__` is controlled by the return value of the `find_loader()` method. However, the subsequent processing of `__path__` is also handled by the functions in `sys.path_hooks`. Thus, when package subcomponents are loaded, the entries in `__path__` are checked by the `handle_url()` function. This causes new instances of `UrlPathFinder` to be created and added to `sys.path_importer_cache`.

One remaining tricky part of the implementation concerns the behavior of the `handle_url()` function and its interaction with the `_get_links()` function used internally. If your implementation of a finder involves the use of other modules (e.g., `urllib.request`), there is a possibility that those modules will attempt to make further imports in the middle of the finder's operation. This can actually cause `handle_url()` and other parts of the finder to get executed in a kind of recursive loop. To account for this possibility, the implementation maintains a cache of created finders (one per URL). This avoids the problem of creating duplicate finders. In addition, the following fragment of code ensures that the finder doesn't respond to any import requests while it's in the process of getting the initial set of links:

```
# Check link cache
if self._links is None:
    self._links = [] # See discussion
    self._links = _get_links(self._baseurl)
```

You may not need this checking in other implementations, but for this example involving URLs, it was required.

Finally, the `invalidate_caches()` method of both finders is a utility method that is supposed to clear internal caches should the source code change. This method is triggered when a user invokes `importlib.invalidate_caches()`. You might use it if you want the URL importers to reread the list of links, possibly for the purpose of being able to access newly added files.

In comparing the two approaches (modifying `sys.meta_path` or using a path hook), it helps to take a high-level view. Importers installed using `sys.meta_path` are free to handle modules in any manner that they wish. For instance, they could load modules out of a database or import them in a manner that is radically different than normal module/package handling. This freedom also means that such importers need to do more bookkeeping and internal management. This explains, for instance, why the implementation of `UrlMetaFinder` needs to do its own caching of links, loaders, and other details. On the other hand, path-based hooks are more narrowly tied to the processing of `sys.path`. Because of the connection to `sys.path`, modules loaded with such extensions will tend to have the same features as normal modules and packages that programmers are used to.

Assuming that your head hasn't completely exploded at this point, a key to understanding and experimenting with this recipe may be the added logging calls. You can enable logging and try experiments such as this:

```
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG)
>>> import urlimport
>>> urlimport.install_path_hook()
DEBUG:urlimport:Installing handle_url
>>> import fib
DEBUG:urlimport:Handle path? /usr/local/lib/python3.3.zip. [No]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'fib'
>>> import sys
>>> sys.path.append('http://localhost:15000')
>>> import fib
DEBUG:urlimport:Handle path? http://localhost:15000. [Yes]
DEBUG:urlimport:Getting links from http://localhost:15000
DEBUG:urlimport:links: {'spam.py', 'fib.py', 'grok'}
DEBUG:urlimport:find_loader: 'fib'
DEBUG:urlimport:find_loader: module 'fib' found
DEBUG:urlimport:loader: reading 'http://localhost:15000/fib.py'
DEBUG:urlimport:loader: 'http://localhost:15000/fib.py' loaded
I'm fib
>>>
```

Last, but not least, spending some time sleeping with [PEP 302](#) and the documentation for `importlib` under your pillow may be advisable.

## 10.12. Patching Modules on Import

### Problem

You want to patch or apply decorators to functions in an existing module. However, you only want to do it if the module actually gets imported and used elsewhere.

### Solution

The essential problem here is that you would like to carry out actions in response to a module being loaded. Perhaps you want to trigger some kind of callback function that would notify you when a module was loaded.

This problem can be solved using the same import hook machinery discussed in [Recipe 10.11](#). Here is a possible solution:

```

# postimport.py

import importlib
import sys
from collections import defaultdict

_post_import_hooks = defaultdict(list)

class PostImportFinder:
    def __init__(self):
        self._skip = set()

    def find_module(self, fullname, path=None):
        if fullname in self._skip:
            return None
        self._skip.add(fullname)
        return PostImportLoader(self)

class PostImportLoader:
    def __init__(self, finder):
        self._finder = finder

    def load_module(self, fullname):
        importlib.import_module(fullname)
        module = sys.modules[fullname]
        for func in _post_import_hooks[fullname]:
            func(module)
        self._finder._skip.remove(fullname)
        return module

def when_imported(fullname):
    def decorate(func):
        if fullname in sys.modules:
            func(sys.modules[fullname])
        else:
            _post_import_hooks[fullname].append(func)
        return func
    return decorate

sys.meta_path.insert(0, PostImportFinder())

```

To use this code, you use the `when_imported()` decorator. For example:

```

>>> from postimport import when_imported
>>> @when_imported('threading')
... def warn_threads(mod):
...     print('Threads? Are you crazy?')
...
>>>
>>> import threading
Threads? Are you crazy?
>>>

```

As a more practical example, maybe you want to apply decorators to existing definitions, such as shown here:

```
from functools import wraps
from postimport import when_imported

def logged(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print('Calling', func.__name__, args, kwargs)
        return func(*args, **kwargs)
    return wrapper

# Example
@when_imported('math')
def add_logging(mod):
    mod.cos = logged(mod.cos)
    mod.sin = logged(mod.sin)
```

## Discussion

This recipe relies on the import hooks that were discussed in [Recipe 10.11](#), with a slight twist.

First, the role of the `@when_imported` decorator is to register handler functions that get triggered on import. The decorator checks `sys.modules` to see if a module was already loaded. If so, the handler is invoked immediately. Otherwise, the handler is added to a list in the `_post_import_hooks` dictionary. The purpose of `_post_import_hooks` is simply to collect all handler objects that have been registered for each module. In principle, more than one handler could be registered for a given module.

To trigger the pending actions in `_post_import_hooks` after module import, the `PostImportFinder` class is installed as the first item in `sys.meta_path`. If you recall from [Recipe 10.11](#), `sys.meta_path` contains a list of finder objects that are consulted in order to locate modules. By installing `PostImportFinder` as the first item, it captures all module imports.

In this recipe, however, the role of `PostImportFinder` is not to load modules, but to trigger actions upon the completion of an import. To do this, the actual import is delegated to the other finders on `sys.meta_path`. Rather than trying to do this directly, the function `imp.import_module()` is called recursively in the `PostImportLoader` class. To avoid getting stuck in an infinite loop, `PostImportFinder` keeps a set of all the modules that are currently in the process of being loaded. If a module name is part of this set, it is simply ignored by `PostImportFinder`. This is what causes the import request to pass to the other finders on `sys.meta_path`.



After a module has been loaded with `imp.import_module()`, all handlers currently registered in `_post_import_hooks` are called with the newly loaded module as an argument. From this point forward, the handlers are free to do what they want with the module.

A major feature of the approach shown in this recipe is that the patching of a module occurs in a seamless fashion, regardless of where or how a module of interest is actually loaded. You simply write a handler function that's decorated with `@when_imported()` and it all just magically works from that point forward.

One caution about this recipe is that it does not work for modules that have been explicitly reloaded using `imp.reload()`. That is, if you reload a previously loaded module, the post import handler function doesn't get triggered again (all the more reason to not use `reload()` in production code). On the other hand, if you delete the module from `sys.modules` and redo the import, you'll see the handler trigger again.

More information about post-import hooks can be found in [PEP 369](#). As of this writing, the PEP has been withdrawn by the author due to it being out of date with the current implementation of the `importlib` module. However, it is easy enough to implement your own solution using this recipe.

## 10.13. Installing Packages Just for Yourself

### Problem

You want to install a third-party package, but you don't have permission to install packages into the system Python. Alternatively, perhaps you just want to install a package for your own use, not all users on the system.

### Solution

Python has a per-user installation directory that's typically located in a directory such as `~/.local/lib/python3.3/site-packages`. To force packages to install in this directory, give the `--user` option to the installation command. For example:

```
python3 setup.py install --user
```

or

```
pip install --user packagename
```

The user *site-packages* directory normally appears before the system *site-packages* directory on `sys.path`. Thus, packages you install using this technique take priority over the packages already installed on the system (although this is not always the case depending on the behavior of third-party package managers, such as `distribute` or `pip`).

## Discussion

Normally, packages get installed into the system-wide *site-packages* directory, which is found in a location such as `/usr/local/lib/python3.3/site-packages`. However, doing so typically requires administrator permissions and use of the `sudo` command. Even if you have permission to execute such a command, using `sudo` to install a new, possibly unproven, package might give you some pause.

Installing packages into the per-user directory is often an effective workaround that allows you to create a custom installation.

As an alternative, you can also create a virtual environment, which is discussed in the next recipe.

## 10.14. Creating a New Python Environment

### Problem

You want to create a new Python environment in which you can install modules and packages. However, you want to do this without installing a new copy of Python or making changes that might affect the system Python installation.

### Solution

You can make a new “virtual” environment using the `pyvenv` command. This command is installed in the same directory as the Python interpreter or possibly in the *Scripts* directory on Windows. Here is an example:

```
bash % pyvenv Spam
bash %
```

The name supplied to `pyvenv` is the name of a directory that will be created. Upon creation, the *Spam* directory will look something like this:

```
bash % cd Spam
bash % ls
bin                include            lib                pyvenv.cfg
bash %
```

In the *bin* directory, you’ll find a Python interpreter that you can use. For example:

```
bash % Spam/bin/python3
Python 3.3.0 (default, Oct  6 2012, 15:45:22)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from pprint import pprint
>>> import sys
>>> pprint(sys.path)
['',
```

```
'/usr/local/lib/python3.3.zip',  
'/usr/local/lib/python3.3',  
'/usr/local/lib/python3.3/plat-darwin',  
'/usr/local/lib/python3.3/lib-dynload',  
'/Users/beazley/Spam/lib/python3.3/site-packages']  
>>>
```

A key feature of this interpreter is that its *site-packages* directory has been set to the newly created environment. Should you decide to install third-party packages, they will be installed here, not in the normal system *site-packages* directory.

## Discussion

The creation of a virtual environment mostly pertains to the installation and management of third-party packages. As you can see in the example, the `sys.path` variable contains directories from the normal system Python, but the *site-packages* directory has been relocated to a new directory.

With a new virtual environment, the next step is often to install a package manager, such as `distribute` or `pip`. When installing such tools and subsequent packages, you just need to make sure you use the interpreter that's part of the virtual environment. This should install the packages into the newly created *site-packages* directory.

Although a virtual environment might look like a copy of the Python installation, it really only consists of a few files and symbolic links. All of the standard library files and interpreter executables come from the original Python installation. Thus, creating such environments is easy, and takes almost no machine resources.

By default, virtual environments are completely clean and contain no third-party additions. If you would like to include already installed packages as part of a virtual environment, create the environment using the `--system-site-packages` option. For example:

```
bash % pyenv --system-site-packages Spam  
bash %
```

More information about `pyenv` and virtual environments can be found in [PEP 405](#).

## 10.15. Distributing Packages

### Problem

You've written a useful library, and you want to be able to give it away to others.

## Solution

If you're going to start giving code away, the first thing to do is to give it a unique name and clean up its directory structure. For example, a typical library package might look something like this:

```
projectname/
  README.txt
  Doc/
    documentation.txt
  projectname/
    __init__.py
    foo.py
    bar.py
    utils/
      __init__.py
      spam.py
      grok.py
  examples/
    helloworld.py
  ...
```

To make the package something that you can distribute, first write a *setup.py* file that looks like this:

```
# setup.py
from distutils.core import setup

setup(name='projectname',
      version='1.0',
      author='Your Name',
      author_email='you@youraddress.com',
      url='http://www.you.com/projectname',
      packages=['projectname', 'projectname.utils'],
    )
```

Next, make a file *MANIFEST.in* that lists various nonsource files that you want to include in your package:

```
# MANIFEST.in
include *.txt
recursive-include examples *
recursive-include Doc *
```

Make sure the *setup.py* and *MANIFEST.in* files appear in the top-level directory of your package. Once you have done this, you should be able to make a source distribution by typing a command such as this:

```
% bash python3 setup.py sdist
```

This will create a file such as *projectname-1.0.zip* or *projectname-1.0.tar.gz*, depending on the platform. If it all works, this file is suitable for giving to others or uploading to the [Python Package Index](#).

## Discussion

For pure Python code, writing a plain *setup.py* file is usually straightforward. One potential gotcha is that you have to manually list every subdirectory that makes up the packages source code. A common mistake is to only list the top-level directory of a package and to forget to include package subcomponents. This is why the specification for packages in *setup.py* includes the list `packages=['projectname', 'projectname.utils']`.

As most Python programmers know, there are many third-party packaging options, including *setuptools*, *distribute*, and so forth. Some of these are replacements for the *distutils* library found in the standard library. Be aware that if you rely on these packages, users may not be able to install your software unless they also install the required package manager first. Because of this, you can almost never go wrong by keeping things as simple as possible. At a bare minimum, make sure your code can be installed using a standard Python 3 installation. Additional features can be supported as an option if additional packages are available.

Packaging and distribution of code involving C extensions can get considerably more complicated. [Chapter 15](#) on C extensions has a few details on this. In particular, see [Recipe 15.2](#).



---

# Network and Web Programming

This chapter is about various topics related to using Python in networked and distributed applications. Topics are split between using Python as a client to access existing services and using Python to implement networked services as a server. Common techniques for writing code involving cooperating or communicating with interpreters are also given.

## 11.1. Interacting with HTTP Services As a Client

### Problem

You need to access various services via HTTP as a client. For example, downloading data or interacting with a REST-based API.

### Solution

For simple things, it's usually easy enough to use the `urllib.request` module. For example, to send a simple HTTP GET request to a remote service, do something like this:

```
from urllib import request, parse

# Base URL being accessed
url = 'http://httpbin.org/get'

# Dictionary of query parameters (if any)
parms = {
    'name1' : 'value1',
    'name2' : 'value2'
}

# Encode the query string
querystring = parse.urlencode(parms)
```

```
# Make a GET request and read the response
u = request.urlopen(url+'?' + querystring)
resp = u.read()
```

If you need to send the query parameters in the request body using a POST method, encode them and supply them as an optional argument to `urlopen()` like this:

```
from urllib import request, parse

# Base URL being accessed
url = 'http://httpbin.org/post'

# Dictionary of query parameters (if any)
parms = {
    'name1' : 'value1',
    'name2' : 'value2'
}

# Encode the query string
querystring = parse.urlencode(parms)

# Make a POST request and read the response
u = request.urlopen(url, querystring.encode('ascii'))
resp = u.read()
```

If you need to supply some custom HTTP headers in the outgoing request such as a change to the user-agent field, make a dictionary containing their value and create a Request instance and pass it to `urlopen()` like this:

```
from urllib import request, parse
...

# Extra headers
headers = {
    'User-agent' : 'none/ofyourbusiness',
    'Spam' : 'Eggs'
}

req = request.Request(url, querystring.encode('ascii'), headers=headers)

# Make a request and read the response
u = request.urlopen(req)
resp = u.read()
```

If your interaction with a service is more complicated than this, you should probably look at the **requests library**. For example, here is equivalent requests code for the preceding operations:

```
import requests

# Base URL being accessed
url = 'http://httpbin.org/post'
```



```

# Dictionary of query parameters (if any)
parms = {
    'name1' : 'value1',
    'name2' : 'value2'
}

# Extra headers
headers = {
    'User-agent' : 'none/ofyourbusiness',
    'Spam' : 'Eggs'
}

resp = requests.post(url, data=parms, headers=headers)

# Decoded text returned by the request
text = resp.text

```

A notable feature of requests is how it returns the resulting response content from a request. As shown, the `resp.text` attribute gives you the Unicode decoded text of a request. However, if you access `resp.content`, you get the raw binary content instead. On the other hand, if you access `resp.json`, then you get the response content interpreted as JSON.

Here is an example of using requests to make a HEAD request and extract a few fields of header data from the response:

```

import requests

resp = requests.head('http://www.python.org/index.html')

status = resp.status_code
last_modified = resp.headers['last-modified']
content_type = resp.headers['content-type']
content_length = resp.headers['content-length']

```

Here is a requests example that executes a login into the Python Package index using basic authentication:

```

import requests

resp = requests.get('http://pypi.python.org/pypi?action=login',
                    auth=('user', 'password'))

```

Here is an example of using requests to pass HTTP cookies from one request to the next:

```

import requests

# First request
resp1 = requests.get(url)
...

```

```
# Second requests with cookies received on first requests
resp2 = requests.get(url, cookies=resp1.cookies)
```

Last, but not least, here is an example of using requests to upload content:

```
import requests
url = 'http://httpbin.org/post'
files = { 'file': ('data.csv', open('data.csv', 'rb')) }

r = requests.post(url, files=files)
```

## Discussion

For really simple HTTP client code, using the built-in `urllib` module is usually fine. However, if you have to do anything other than simple GET or POST requests, you really can't rely on its functionality. This is where a third-party module, such as `requests`, comes in handy.

For example, if you decided to stick entirely with the standard library instead of a library like `requests`, you might have to implement your code using the low-level `http.client` module instead. For example, this code shows how to execute a HEAD request:

```
from http.client import HTTPConnection
from urllib import parse

c = HTTPConnection('www.python.org', 80)
c.request('HEAD', '/index.html')
resp = c.getresponse()

print('Status', resp.status)
for name, value in resp.getheaders():
    print(name, value)
```

Similarly, if you have to write code involving proxies, authentication, cookies, and other details, using `urllib` is awkward and verbose. For example, here is a sample of code that authenticates to the Python package index:

```
import urllib.request

auth = urllib.request.HTTPBasicAuthHandler()
auth.add_password('pypi', 'http://pypi.python.org', 'username', 'password')
opener = urllib.request.build_opener(auth)

r = urllib.request.Request('http://pypi.python.org/pypi?action=login')
u = opener.open(r)
resp = u.read()

# From here. You can access more pages using opener
...
```

Frankly, all of this is much easier in `requests`.

Testing HTTP client code during development can often be frustrating because of all the tricky details you need to worry about (e.g., cookies, authentication, headers, encodings, etc.). To do this, consider using the [httpbin service](http://httpbin.org). This site receives requests and then echoes information back to you in the form a JSON response. Here is an interactive example:

```
>>> import requests
>>> r = requests.get('http://httpbin.org/get?name=Dave&n=37',
...                 headers = { 'User-agent': 'goaway/1.0' })
>>> resp = r.json
>>> resp['headers']
{'User-Agent': 'goaway/1.0', 'Content-Length': '', 'Content-Type': '',
 'Accept-Encoding': 'gzip, deflate, compress', 'Connection':
 'keep-alive', 'Host': 'httpbin.org', 'Accept': '/*/*'}
>>> resp['args']
{'name': 'Dave', 'n': '37'}
>>>
```

Working with a site such as [httpbin.org](http://httpbin.org) is often preferable to experimenting with a real site—especially if there’s a risk it might shut down your account after three failed login attempts (i.e., don’t try to learn how to write an HTTP authentication client by logging into your bank).

Although it’s not discussed here, `requests` provides support for many more advanced HTTP-client protocols, such as OAuth. The [requests documentation](#) is excellent (and frankly better than anything that could be provided in this short space). Go there for more information.

## 11.2. Creating a TCP Server

### Problem

You want to implement a server that communicates with clients using the TCP Internet protocol.

### Solution

An easy way to create a TCP server is to use the `socketserver` library. For example, here is a simple echo server:

```
from socketserver import BaseRequestHandler, TCPServer

class EchoHandler(BaseRequestHandler):
    def handle(self):
        print('Got connection from', self.client_address)
        while True:
```

```

        msg = self.request.recv(8192)
        if not msg:
            break
        self.request.send(msg)

if __name__ == '__main__':
    serv = TCPServer(('', 20000), EchoHandler)
    serv.serve_forever()

```

In this code, you define a special handler class that implements a `handle()` method for servicing client connections. The `request` attribute is the underlying client socket and `client_address` has client address.

To test the server, run it and then open a separate Python process that connects to it:

```

>>> from socket import socket, AF_INET, SOCK_STREAM
>>> s = socket(AF_INET, SOCK_STREAM)
>>> s.connect(('localhost', 20000))
>>> s.send(b'Hello')
5
>>> s.recv(8192)
b'Hello'
>>>

```

In many cases, it may be easier to define a slightly different kind of handler. Here is an example that uses the `StreamRequestHandler` base class to put a file-like interface on the underlying socket:

```

from socketserver import StreamRequestHandler, TCPServer

class EchoHandler(StreamRequestHandler):
    def handle(self):
        print('Got connection from', self.client_address)
        # self.rfile is a file-like object for reading
        for line in self.rfile:
            # self.wfile is a file-like object for writing
            self.wfile.write(line)

if __name__ == '__main__':
    serv = TCPServer(('', 20000), EchoHandler)
    serv.serve_forever()

```

## Discussion

`socketserver` makes it relatively easy to create simple TCP servers. However, you should be aware that, by default, the servers are single threaded and can only serve one client at a time. If you want to handle multiple clients, either instantiate a `ForkingTCPServer` or `ThreadingTCPServer` object instead. For example:

```

from socketserver import ThreadingTCPServer
...

```

```

if __name__ == '__main__':
    serv = ThreadingTCPServer(('', 20000), EchoHandler)
    serv.serve_forever()

```

One issue with forking and threaded servers is that they spawn a new process or thread on each client connection. There is no upper bound on the number of allowed clients, so a malicious hacker could potentially launch a large number of simultaneous connections in an effort to make your server explode.

If this is a concern, you can create a pre-allocated pool of worker threads or processes. To do this, you create an instance of a normal nonthreaded server, but then launch the `serve_forever()` method in a pool of multiple threads. For example:

```

...
if __name__ == '__main__':
    from threading import Thread
    NWORKERS = 16
    serv = TCPServer(('', 20000), EchoHandler)
    for n in range(NWORKERS):
        t = Thread(target=serv.serve_forever)
        t.daemon = True
        t.start()
    serv.serve_forever()

```

Normally, a `TCPServer` binds and activates the underlying socket upon instantiation. However, sometimes you might want to adjust the underlying socket by setting options. To do this, supply the `bind_and_activate=False` argument, like this:

```

if __name__ == '__main__':
    serv = TCPServer(('', 20000), EchoHandler, bind_and_activate=False)
    # Set up various socket options
    serv.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
    # Bind and activate
    serv.server_bind()
    serv.server_activate()
    serv.serve_forever()

```

The socket option shown is actually a very common setting that allows the server to rebind to a previously used port number. It's actually so common that it's a class variable that can be set on `TCPServer`. Set it before instantiating the server, as shown in this example:

```

...
if __name__ == '__main__':
    TCPServer.allow_reuse_address = True
    serv = TCPServer(('', 20000), EchoHandler)
    serv.serve_forever()

```

In the solution, two different handler base classes were shown (`BaseRequestHandler` and `StreamRequestHandler`). The `StreamRequestHandler` class is actually a bit more

flexible, and supports some features that can be enabled through the specification of additional class variables. For example:

```
import socket

class EchoHandler(StreamRequestHandler):
    # Optional settings (defaults shown)
    timeout = 5          # Timeout on all socket operations
    rbufsize = -1        # Read buffer size
    wbufsize = 0         # Write buffer size
    disable_nagle_algorithm = False # Sets TCP_NODELAY socket option
    def handle(self):
        print('Got connection from', self.client_address)
        try:
            for line in self.rfile:
                # self.wfile is a file-like object for writing
                self.wfile.write(line)
        except socket.timeout:
            print('Timed out!')
```

Finally, it should be noted that most of Python's higher-level networking modules (e.g., HTTP, XML-RPC, etc.) are built on top of the socketserver functionality. That said, it is also not difficult to implement servers directly using the socket library as well. Here is a simple example of directly programming a server with Sockets:

```
from socket import socket, AF_INET, SOCK_STREAM

def echo_handler(address, client_sock):
    print('Got connection from {}'.format(address))
    while True:
        msg = client_sock.recv(8192)
        if not msg:
            break
        client_sock.sendall(msg)
    client_sock.close()

def echo_server(address, backlog=5):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(address)
    sock.listen(backlog)
    while True:
        client_sock, client_addr = sock.accept()
        echo_handler(client_addr, client_sock)

if __name__ == '__main__':
    echo_server('', 20000)
```

## 11.3. Creating a UDP Server

### Problem

You want to implement a server that communicates with clients using the UDP Internet protocol.

### Solution

As with TCP, UDP servers are also easy to create using the `socketserver` library. For example, here is a simple time server:

```
from socketserver import BaseRequestHandler, UDPServer
import time

class TimeHandler(BaseRequestHandler):
    def handle(self):
        print('Got connection from', self.client_address)
        # Get message and client socket
        msg, sock = self.request
        resp = time.ctime()
        sock.sendto(resp.encode('ascii'), self.client_address)

if __name__ == '__main__':
    serv = UDPServer(('', 20000), TimeHandler)
    serv.serve_forever()
```

As before, you define a special handler class that implements a `handle()` method for servicing client connections. The request attribute is a tuple that contains the incoming datagram and underlying socket object for the server. The `client_address` contains the client address.

To test the server, run it and then open a separate Python process that sends messages to it:

```
>>> from socket import socket, AF_INET, SOCK_DGRAM
>>> s = socket(AF_INET, SOCK_DGRAM)
>>> s.sendto(b'', ('localhost', 20000))
0
>>> s.recvfrom(8192)
(b'Wed Aug 15 20:35:08 2012', ('127.0.0.1', 20000))
>>>
```

### Discussion

A typical UDP server receives an incoming datagram (message) along with a client address. If the server is to respond, it sends a datagram back to the client. For transmission of datagrams, you should use the `sendto()` and `recvfrom()` methods of a

socket. Although the traditional `send()` and `recv()` methods also might work, the former two methods are more commonly used with UDP communication.

Given that there is no underlying connection, UDP servers are often much easier to write than a TCP server. However, UDP is also inherently unreliable (e.g., no “connection” is established and messages might be lost). Thus, it would be up to you to figure out how to deal with lost messages. That’s a topic beyond the scope of this book, but typically you might need to introduce sequence numbers, retries, timeouts, and other mechanisms to ensure reliability if it matters for your application. UDP is often used in cases where the requirement of reliable delivery can be relaxed. For instance, in real-time applications such as multimedia streaming and games where there is simply no option to go back in time and recover a lost packet (the program simply skips it and keeps moving forward).

The `UDPServer` class is single threaded, which means that only one request can be serviced at a time. In practice, this is less of an issue with UDP than with TCP connections. However, should you want concurrent operation, instantiate a `ForkingUDPServer` or `ThreadingUDPServer` object instead:

```
from socketserver import ThreadingUDPServer
...
if __name__ == '__main__':
    serv = ThreadingUDPServer(('', 20000), TimeHandler)
    serv.serve_forever()
```

Implementing a UDP server directly using sockets is also not difficult. Here is an example:

```
from socket import socket, AF_INET, SOCK_DGRAM
import time

def time_server(address):
    sock = socket(AF_INET, SOCK_DGRAM)
    sock.bind(address)
    while True:
        msg, addr = sock.recvfrom(8192)
        print('Got message from', addr)
        resp = time.ctime()
        sock.sendto(resp.encode('ascii'), addr)

if __name__ == '__main__':
    time_server(('', 20000))
```



## 11.4. Generating a Range of IP Addresses from a CIDR Address

### Problem

You have a CIDR network address such as “123.45.67.89/27,” and you want to generate a range of all the IP addresses that it represents (e.g., “123.45.67.64,” “123.45.67.65,” ..., “123.45.67.95”).

### Solution

The `ipaddress` module can be easily used to perform such calculations. For example:

```
>>> import ipaddress
>>> net = ipaddress.ip_network('123.45.67.64/27')
>>> net
IPv4Network('123.45.67.64/27')
>>> for a in net:
...     print(a)
...
123.45.67.64
123.45.67.65
123.45.67.66
123.45.67.67
123.45.67.68
...
123.45.67.95
>>>

>>> net6 = ipaddress.ip_network('12:3456:78:90ab:cd:ef01:23:30/125')
>>> net6
IPv6Network('12:3456:78:90ab:cd:ef01:23:30/125')
>>> for a in net6:
...     print(a)
...
12:3456:78:90ab:cd:ef01:23:30
12:3456:78:90ab:cd:ef01:23:31
12:3456:78:90ab:cd:ef01:23:32
12:3456:78:90ab:cd:ef01:23:33
12:3456:78:90ab:cd:ef01:23:34
12:3456:78:90ab:cd:ef01:23:35
12:3456:78:90ab:cd:ef01:23:36
12:3456:78:90ab:cd:ef01:23:37
>>>
```

Network objects also allow indexing like arrays. For example:

```
>>> net.num_addresses
32
>>> net[0]
```

```

IPv4Address('123.45.67.64')
>>> net[1]
IPv4Address('123.45.67.65')
>>> net[-1]
IPv4Address('123.45.67.95')
>>> net[-2]
IPv4Address('123.45.67.94')
>>>

```

In addition, you can perform operations such as a check for network membership:

```

>>> a = ipaddress.ip_address('123.45.67.69')
>>> a in net
True
>>> b = ipaddress.ip_address('123.45.67.123')
>>> b in net
False
>>>

```

An IP address and network address can be specified together as an IP interface. For example:

```

>>> inet = ipaddress.ip_interface('123.45.67.73/27')
>>> inet.network
IPv4Network('123.45.67.64/27')
>>> inet.ip
IPv4Address('123.45.67.73')
>>>

```

## Discussion

The `ipaddress` module has classes for representing IP addresses, networks, and interfaces. This can be especially useful if you want to write code that needs to manipulate network addresses in some way (e.g., parsing, printing, validating, etc.).

Be aware that there is only limited interaction between the `ipaddress` module and other network-related modules, such as the `socket` library. In particular, it is usually not possible to use an instance of `IPv4Address` as a substitute for address string. Instead, you have to explicitly convert it using `str()` first. For example:

```

>>> a = ipaddress.ip_address('127.0.0.1')
>>> from socket import socket, AF_INET, SOCK_STREAM
>>> s = socket(AF_INET, SOCK_STREAM)
>>> s.connect((a, 8080))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'IPv4Address' object to str implicitly
>>> s.connect((str(a), 8080))
>>>

```

See “[An Introduction to the ipaddress Module](#)” for more information and advanced usage.

## 11.5. Creating a Simple REST-Based Interface

### Problem

You want to be able to control or interact with your program remotely over the network using a simple REST-based interface. However, you don't want to do it by installing a full-fledged web programming framework.

### Solution

One of the easiest ways to build REST-based interfaces is to create a tiny library based on the WSGI standard, as described in [PEP 3333](#). Here is an example:

```
# resty.py

import cgi

def notfound_404(envIRON, start_response):
    start_response('404 Not Found', [ ('Content-type', 'text/plain') ])
    return [b'Not Found']

class PathDispatcher:
    def __init__(self):
        self.pathmap = { }

    def __call__(self, environ, start_response):
        path = environ['PATH_INFO']
        params = cgi.FieldStorage(environ['wsgi.input'],
                                  environ=environ)
        method = environ['REQUEST_METHOD'].lower()
        environ['params'] = { key: params.getvalue(key) for key in params }
        handler = self.pathmap.get((method,path), notfound_404)
        return handler(environ, start_response)

    def register(self, method, path, function):
        self.pathmap[method.lower(), path] = function
        return function
```

To use this dispatcher, you simply write different handlers, such as the following:

```
import time

_hello_resp = '''\
<html>
  <head>
    <title>Hello {name}</title>
  </head>
  <body>
    <h1>Hello {name}!</h1>
  </body>
</html>'''
```

```

def hello_world(envIRON, start_response):
    start_response('200 OK', [ ('Content-type', 'text/html') ])
    params = environ['params']
    resp = _hello_resp.format(name=params.get('name'))
    yield resp.encode('utf-8')

_localtime_resp = '''\
<?xml version="1.0"?>
<time>
  <year>{t.tm_year}</year>
  <month>{t.tm_mon}</month>
  <day>{t.tm_mday}</day>
  <hour>{t.tm_hour}</hour>
  <minute>{t.tm_min}</minute>
  <second>{t.tm_sec}</second>
</time>'''

def localtime(envIRON, start_response):
    start_response('200 OK', [ ('Content-type', 'application/xml') ])
    resp = _localtime_resp.format(t=time.localtime())
    yield resp.encode('utf-8')

if __name__ == '__main__':
    from resty import PathDispatcher
    from wsgiref.simple_server import make_server

    # Create the dispatcher and register functions
    dispatcher = PathDispatcher()
    dispatcher.register('GET', '/hello', hello_world)
    dispatcher.register('GET', '/localtime', localtime)

    # Launch a basic server
    httpd = make_server('', 8080, dispatcher)
    print('Serving on port 8080...')
    httpd.serve_forever()

```

To test your server, you can interact with it using a browser or `urllib`. For example:

```

>>> u = urlopen('http://localhost:8080/hello?name=Guido')
>>> print(u.read().decode('utf-8'))
<html>
  <head>
    <title>Hello Guido</title>
  </head>
  <body>
    <h1>Hello Guido!</h1>
  </body>
</html>
>>> u = urlopen('http://localhost:8080/localtime')
>>> print(u.read().decode('utf-8'))
<?xml version="1.0"?>
<time>

```

```

<year>2012</year>
<month>11</month>
<day>24</day>
<hour>14</hour>
<minute>49</minute>
<second>17</second>
</time>
>>>

```

## Discussion

In REST-based interfaces, you are typically writing programs that respond to common HTTP requests. However, unlike a full-fledged website, you're often just pushing data around. This data might be encoded in a variety of standard formats such as XML, JSON, or CSV. Although it seems minimal, providing an API in this manner can be a very useful thing for a wide variety of applications.

For example, long-running programs might use a REST API to implement monitoring or diagnostics. Big data applications can use REST to build a query/data extraction system. REST can even be used to control hardware devices, such as robots, sensors, mills, or lightbulbs. What's more, REST APIs are well supported by various client-side programming environments, such as Javascript, Android, iOS, and so forth. Thus, having such an interface can be a way to encourage the development of more complex applications that interface with your code.

For implementing a simple REST interface, it is often easy enough to base your code on the Python WSGI standard. WSGI is supported by the standard library, but also by most third-party web frameworks. Thus, if you use it, there is a lot of flexibility in how your code might be used later.

In WSGI, you simply implement applications in the form of a callable that accepts this calling convention:

```

import cgi

def wsgi_app(environ, start_response):
    ...

```

The `environ` argument is a dictionary that contains values inspired by the CGI interface provided by various web servers such as Apache [see [Internet RFC 3875](#)]. To extract different fields, you would write code like this:

```

def wsgi_app(environ, start_response):
    method = environ['REQUEST_METHOD']
    path = environ['PATH_INFO']
    # Parse the query parameters
    params = cgi.FieldStorage(environ['wsgi.input'], environ=environ)
    ...

```

A few common values are shown here. `environ['REQUEST_METHOD']` is the type of request (e.g., GET, POST, HEAD, etc.). `environ['PATH_INFO']` is the path or the resource being requested. The call to `cgi.FieldStorage()` extracts supplied query parameters from the request and puts them into a dictionary-like object for later use.

The `start_response` argument is a function that must be called to initiate a response. The first argument is the resulting HTTP status. The second argument is a list of (name, value) tuples that make up the HTTP headers of the response. For example:

```
def wsgi_app(environ, start_response):
    ...
    start_response('200 OK', [('Content-type', 'text/plain')])
```

To return data, an WSGI application must return a sequence of byte strings. This can be done using a list like this:

```
def wsgi_app(environ, start_response):
    ...
    start_response('200 OK', [('Content-type', 'text/plain')])
    resp = []
    resp.append(b'Hello World\n')
    resp.append(b'Goodbye!\n')
    return resp
```

Alternatively, you can use `yield`:

```
def wsgi_app(environ, start_response):
    ...
    start_response('200 OK', [('Content-type', 'text/plain')])
    yield b'Hello World\n'
    yield b'Goodbye!\n'
```

It's important to emphasize that byte strings must be used in the result. If the response consists of text, it will need to be encoded into bytes first. Of course, there is no requirement that the returned value be text—you could easily write an application function that creates images.

Although WSGI applications are commonly defined as a function, as shown, an instance may also be used as long as it implements a suitable `__call__()` method. For example:

```
class WSGIApplication:
    def __init__(self):
        ...
    def __call__(self, environ, start_response)
        ...
```

This technique has been used to create the `PathDispatcher` class in the recipe. The dispatcher does nothing more than manage a dictionary mapping (method, path) pairs to handler functions. When a request arrives, the method and path are extracted and used to dispatch to a handler. In addition, any query variables are parsed and put into

a dictionary that is stored as `environ[ 'params' ]` (this latter step is so common, it makes a lot of sense to simply do it in the dispatcher in order to avoid a lot of replicated code).

To use the dispatcher, you simply create an instance and register various WSGI-style application functions with it, as shown in the recipe. Writing these functions should be extremely straightforward, as you follow the rules concerning the `start_response()` function and produce output as byte strings.

One thing to consider when writing such functions is the careful use of string templates. Nobody likes to work with code that is a tangled mess of `print()` functions, XML, and various formatting operations. In the solution, triple-quoted string templates are being defined and used internally. This particular approach makes it easier to change the format of the output later (just change the template as opposed to any of the code that uses it).

Finally, an important part of using WSGI is that nothing in the implementation is specific to a particular web server. That is actually the whole idea—since the standard is server and framework neutral, you should be able to plug your application into a wide variety of servers. In the recipe, the following code is used for testing:

```
if __name__ == '__main__':
    from wsgiref.simple_server import make_server

    # Create the dispatcher and register functions
    dispatcher = PathDispatcher()
    ...

    # Launch a basic server
    httpd = make_server('', 8080, dispatcher)
    print('Serving on port 8080...')
    httpd.serve_forever()
```

This will create a simple server that you can use to see if your implementation works. Later on, when you're ready to scale things up to a larger level, you will change this code to work with a particular server.

WSGI is an intentionally minimal specification. As such, it doesn't provide any support for more advanced concepts such as authentication, cookies, redirection, and so forth. These are not hard to implement yourself. However, if you want just a bit more support, you might consider third-party libraries, such as [WebOb](#) or [Paste](#).

## 11.6. Implementing a Simple Remote Procedure Call with XML-RPC

### Problem

You want an easy way to execute functions or methods in Python programs running on remote machines.

### Solution

Perhaps the easiest way to implement a simple remote procedure call mechanism is to use XML-RPC. Here is an example of a simple server that implements a simple key-value store:

```
from xmlrpc.server import SimpleXMLRPCServer

class KeyValueServer:
    _rpc_methods_ = ['get', 'set', 'delete', 'exists', 'keys']
    def __init__(self, address):
        self._data = {}
        self._serv = SimpleXMLRPCServer(address, allow_none=True)
        for name in self._rpc_methods_:
            self._serv.register_function(getattr(self, name))

    def get(self, name):
        return self._data[name]

    def set(self, name, value):
        self._data[name] = value

    def delete(self, name):
        del self._data[name]

    def exists(self, name):
        return name in self._data

    def keys(self):
        return list(self._data)

    def serve_forever(self):
        self._serv.serve_forever()

# Example
if __name__ == '__main__':
    kvserv = KeyValueServer((' ', 15000))
    kvserv.serve_forever()
```

Here is how you would access the server remotely from a client:



```

>>> from xmlrpc.client import ServerProxy
>>> s = ServerProxy('http://localhost:15000', allow_none=True)
>>> s.set('foo', 'bar')
>>> s.set('spam', [1, 2, 3])
>>> s.keys()
['spam', 'foo']
>>> s.get('foo')
'bar'
>>> s.get('spam')
[1, 2, 3]
>>> s.delete('spam')
>>> s.exists('spam')
False
>>>

```

## Discussion

XML-RPC can be an extremely easy way to set up a simple remote procedure call service. All you need to do is create a server instance, register functions with it using the `register_function()` method, and then launch it using the `serve_forever()` method. This recipe packages it up into a class to put all of the code together, but there is no such requirement. For example, you could create a server by trying something like this:

```

from xmlrpc.server import SimpleXMLRPCServer
def add(x,y):
    return x+y

serv = SimpleXMLRPCServer(('', 15000))
serv.register_function(add)
serv.serve_forever()

```

Functions exposed via XML-RPC only work with certain kinds of data such as strings, numbers, lists, and dictionaries. For everything else, some study is required. For instance, if you pass an instance through XML-RPC, only its instance dictionary is handled:

```

>>> class Point:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...
>>> p = Point(2, 3)
>>> s.set('foo', p)
>>> s.get('foo')
{'x': 2, 'y': 3}
>>>

```

Similarly, handling of binary data is a bit different than you expect:

```

>>> s.set('foo', b'Hello World')
>>> s.get('foo')
<xmlrpc.client.Binary object at 0x10131d410>

```

```
>>> _.data
b'Hello World'
>>>
```

As a general rule, you probably shouldn't expose an XML-RPC service to the rest of the world as a public API. It often works best on internal networks where you might want to write simple distributed programs involving a few different machines.

A downside to XML-RPC is its performance. The `SimpleXMLRPCServer` implementation is only single threaded, and wouldn't be appropriate for scaling a large application, although it can be made to run multithreaded, as shown in [Recipe 11.2](#). Also, since XML-RPC serializes all data as XML, it's inherently slower than other approaches. However, one benefit of this encoding is that it's understood by a variety of other programming languages. By using it, clients written in languages other than Python will be able to access your service.

Despite its limitations, XML-RPC is worth knowing about if you ever have the need to make a quick and dirty remote procedure call system. Oftentimes, the simple solution is good enough.

## 11.7. Communicating Simply Between Interpreters

### Problem

You are running multiple instances of the Python interpreter, possibly on different machines, and you would like to exchange data between interpreters using messages.

### Solution

It is easy to communicate between interpreters if you use the `multiprocessing.connection` module. Here is a simple example of writing an echo server:

```
from multiprocessing.connection import Listener
import traceback

def echo_client(conn):
    try:
        while True:
            msg = conn.recv()
            conn.send(msg)
    except EOFError:
        print('Connection closed')

def echo_server(address, authkey):
    serv = Listener(address, authkey=authkey)
    while True:
        try:
            client = serv.accept()
```

```

        echo_client(client)
    except Exception:
        traceback.print_exc()

    echo_server(('', 25000), authkey=b'peekaboo')

```

Here is a simple example of a client connecting to the server and sending various messages:

```

>>> from multiprocessing.connection import Client
>>> c = Client(('localhost', 25000), authkey=b'peekaboo')
>>> c.send('hello')
>>> c.recv()
'hello'
>>> c.send(42)
>>> c.recv()
42
>>> c.send([1, 2, 3, 4, 5])
>>> c.recv()
[1, 2, 3, 4, 5]
>>>

```

Unlike a low-level socket, messages are kept intact (each object sent using `send()` is received in its entirety with `recv()`). In addition, objects are serialized using `pickle`. So, any object compatible with `pickle` can be sent or received over the connection.

## Discussion

There are many packages and libraries related to implementing various forms of message passing, such as ZeroMQ, Celery, and so forth. As an alternative, you might also be inclined to implement a message layer on top of low-level sockets. However, sometimes you just want a simple solution. The `multiprocessing.connection` library is just that—using a few simple primitives, you can easily connect interpreters together and have them exchange messages.

If you know that the interpreters are going to be running on the same machine, you can use alternative forms of networking, such as UNIX domain sockets or Windows named pipes. To create a connection using a UNIX domain socket, simply change the address to a filename such as this:

```
s = Listener('/tmp/myconn', authkey=b'peekaboo')
```

To create a connection using a Windows named pipe, use a filename such as this:

```
s = Listener(r'\\.\pipe\myconn', authkey=b'peekaboo')
```

As a general rule, you would not be using `multiprocessing` to implement public-facing services. The `authkey` parameter to `Client()` and `Listener()` is there to help authenticate the end points of the connection. Connection attempts with a bad key raise an exception. In addition, the module is probably best suited for long-running connections

(not a large number of short connections). For example, two interpreters might establish a connection at startup and keep the connection active for the entire duration of a problem.

Don't use multiprocessing if you need more low-level control over aspects of the connection. For example, if you needed to support timeouts, nonblocking I/O, or anything similar, you're probably better off using a different library or implementing such features on top of sockets instead.

## 11.8. Implementing Remote Procedure Calls

### Problem

You want to implement simple remote procedure call (RPC) on top of a message passing layer, such as sockets, multiprocessing connections, or ZeroMQ.

### Solution

RPC is easy to implement by encoding function requests, arguments, and return values using pickle, and passing the pickled byte strings between interpreters. Here is an example of a simple RPC handler that could be incorporated into a server:

```
# rpcserver.py

import pickle
class RPCHandler:
    def __init__(self):
        self._functions = { }

    def register_function(self, func):
        self._functions[func.__name__] = func

    def handle_connection(self, connection):
        try:
            while True:
                # Receive a message
                func_name, args, kwargs = pickle.loads(connection.recv())
                # Run the RPC and send a response
                try:
                    r = self._functions[func_name](*args,**kwargs)
                    connection.send(pickle.dumps(r))
                except Exception as e:
                    connection.send(pickle.dumps(e))
        except EOFError:
            pass
```

To use this handler, you need to add it into a messaging server. There are many possible choices, but the multiprocessing library provides a simple option. Here is an example RPC server:

```
from multiprocessing.connection import Listener
from threading import Thread

def rpc_server(handler, address, authkey):
    sock = Listener(address, authkey=authkey)
    while True:
        client = sock.accept()
        t = Thread(target=handler.handle_connection, args=(client,))
        t.daemon = True
        t.start()

# Some remote functions
def add(x, y):
    return x + y

def sub(x, y):
    return x - y

# Register with a handler
handler = RPCHandler()
handler.register_function(add)
handler.register_function(sub)

# Run the server
rpc_server(handler, ('localhost', 17000), authkey=b'peekaboo')
```

To access the server from a remote client, you need to create a corresponding RPC proxy class that forwards requests. For example:

```
import pickle

class RPCProxy:
    def __init__(self, connection):
        self._connection = connection
    def __getattr__(self, name):
        def do_rpc(*args, **kwargs):
            self._connection.send(pickle.dumps((name, args, kwargs)))
            result = pickle.loads(self._connection.recv())
            if isinstance(result, Exception):
                raise result
            return result
        return do_rpc
```

To use the proxy, you wrap it around a connection to the server. For example:

```
>>> from multiprocessing.connection import Client
>>> c = Client(('localhost', 17000), authkey=b'peekaboo')
>>> proxy = RPCProxy(c)
>>> proxy.add(2, 3)
```

```

5
>>> proxy.sub(2, 3)
-1
>>> proxy.sub([1, 2], 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "rpcserver.py", line 37, in do_rpc
        raise result
TypeError: unsupported operand type(s) for -: 'list' and 'int'
>>>

```

It should be noted that many messaging layers (such as multiprocessing) already serialize data using pickle. If this is the case, the `pickle.dumps()` and `pickle.loads()` calls can be eliminated.

## Discussion

The general idea of the `RPCHandler` and `RPCProxy` classes is relatively simple. If a client wants to call a remote function, such as `foo(1, 2, z=3)`, the proxy class creates a tuple `('foo', (1, 2), {'z': 3})` that contains the function name and arguments. This tuple is pickled and sent over the connection. This is performed in the `do_rpc()` closure that's returned by the `__getattr__()` method of `RPCProxy`. The server receives and unpickles the message, looks up the function name to see if it's registered, and executes it with the given arguments. The result (or exception) is then pickled and sent back.

As shown, the example relies on multiprocessing for communication. However, this approach could be made to work with just about any other messaging system. For example, if you want to implement RPC over ZeroMQ, just replace the connection objects with an appropriate ZeroMQ socket object.

Given the reliance on pickle, security is a major concern (because a clever hacker can create messages that make arbitrary functions execute during unpickling). In particular, you should never allow RPC from untrusted or unauthenticated clients. In particular, you definitely don't want to allow access from just any machine on the Internet—this should really only be used internally, behind a firewall, and not exposed to the rest of the world.

As an alternative to pickle, you might consider the use of JSON, XML, or some other data encoding for serialization. For example, this recipe is fairly easy to adapt to JSON encoding if you simply replace `pickle.loads()` and `pickle.dumps()` with `json.loads()` and `json.dumps()`. For example:

```

# jsonrpcserver.py
import json

class RPCHandler:
    def __init__(self):
        self._functions = { }

```

```

def register_function(self, func):
    self._functions[func.__name__] = func

def handle_connection(self, connection):
    try:
        while True:
            # Receive a message
            func_name, args, kwargs = json.loads(connection.recv())
            # Run the RPC and send a response
            try:
                r = self._functions[func_name](*args,**kwargs)
                connection.send(json.dumps(r))
            except Exception as e:
                connection.send(json.dumps(str(e)))
    except EOFError:
        pass

# jsonrpcclient.py
import json

class RPCProxy:
    def __init__(self, connection):
        self._connection = connection
    def __getattr__(self, name):
        def do_rpc(*args, **kwargs):
            self._connection.send(json.dumps((name, args, kwargs)))
            result = json.loads(self._connection.recv())
            return result
        return do_rpc

```

One complicated factor in implementing RPC is how to handle exceptions. At the very least, the server shouldn't crash if an exception is raised by a method. However, the means by which the exception gets reported back to the client requires some study. If you're using pickle, exception instances can often be serialized and reraised in the client. If you're using some other protocol, you might have to think of an alternative approach. At the very least, you would probably want to return the exception string in the response. This is the approach taken in the JSON example.

For another example of an RPC implementation, it can be useful to look at the implementation of the SimpleXMLRPCServer and ServerProxy classes used in XML-RPC, as described in [Recipe 11.6](#).

## 11.9. Authenticating Clients Simply

### Problem

You want a simple way to authenticate the clients connecting to servers in a distributed system, but don't need the complexity of something like SSL.

## Solution

Simple but effective authentication can be performed by implementing a connection handshake using the `hmac` module. Here is sample code:

```
import hmac
import os

def client_authenticate(connection, secret_key):
    """
    Authenticate client to a remote service.
    connection represents a network connection.
    secret_key is a key known only to both client/server.
    """
    message = connection.recv(32)
    hash = hmac.new(secret_key, message)
    digest = hash.digest()
    connection.send(digest)

def server_authenticate(connection, secret_key):
    """
    Request client authentication.
    """
    message = os.urandom(32)
    connection.send(message)
    hash = hmac.new(secret_key, message)
    digest = hash.digest()
    response = connection.recv(len(digest))
    return hmac.compare_digest(digest, response)
```

The general idea is that upon connection, the server presents the client with a message of random bytes (returned by `os.urandom()`, in this case). The client and server both compute a cryptographic hash of the random data using `hmac` and a secret key known only to both ends. The client sends its computed digest back to the server, where it is compared and used to decide whether or not to accept or reject the connection.

Comparison of resulting digests should be performed using the `hmac.compare_digest()` function. This function has been written in a way that avoids timing-analysis-based attacks and should be used instead of a normal comparison operator (`==`).

To use these functions, you would incorporate them into existing networking or messaging code. For example, with sockets, the server code might look something like this:

```
from socket import socket, AF_INET, SOCK_STREAM

secret_key = b'peekaboo'
def echo_handler(client_sock):
    if not server_authenticate(client_sock, secret_key):
        client_sock.close()
    return
while True:
```



```

msg = client_sock.recv(8192)
if not msg:
    break
client_sock.sendall(msg)

def echo_server(address):
    s = socket(AF_INET, SOCK_STREAM)
    s.bind(address)
    s.listen(5)
    while True:
        c,a = s.accept()
        echo_handler(c)

echo_server(('', 18000))

```

Within a client, you would do this:

```

from socket import socket, AF_INET, SOCK_STREAM

secret_key = b'peekaboo'

s = socket(AF_INET, SOCK_STREAM)
s.connect(('localhost', 18000))
client_authenticate(s, secret_key)
s.send(b'Hello World')
resp = s.recv(1024)
...

```

## Discussion

A common use of `hmac` authentication is in internal messaging systems and interprocess communication. For example, if you are writing a system that involves multiple processes communicating across a cluster of machines, you can use this approach to make sure that only allowed processes are allowed to connect to one another. In fact, HMAC-based authentication is used internally by the `multiprocessing` library when it sets up communication with subprocesses.

It's important to stress that authenticating a connection is not the same as encryption. Subsequent communication on an authenticated connection is sent in the clear, and would be visible to anyone inclined to sniff the traffic (although the secret key known to both sides is never transmitted).

The authentication algorithm used by `hmac` is based on cryptographic hashing functions, such as MD5 and SHA-1, and is described in detail in [IETF RFC 2104](#).

## 11.10. Adding SSL to Network Services

### Problem

You want to implement a network service involving sockets where servers and clients authenticate themselves and encrypt the transmitted data using SSL.

### Solution

The `ssl` module provides support for adding SSL to low-level socket connections. In particular, the `ssl.wrap_socket()` function takes an existing socket and wraps an SSL layer around it. For example, here's an example of a simple echo server that presents a server certificate to connecting clients:

```
from socket import socket, AF_INET, SOCK_STREAM
import ssl

KEYFILE = 'server_key.pem' # Private key of the server
CERTFILE = 'server_cert.pem' # Server certificate (given to client)

def echo_client(s):
    while True:
        data = s.recv(8192)
        if data == b'':
            break
        s.send(data)
    s.close()
    print('Connection closed')

def echo_server(address):
    s = socket(AF_INET, SOCK_STREAM)
    s.bind(address)
    s.listen(1)

    # Wrap with an SSL layer requiring client certs
    s_ssl = ssl.wrap_socket(s,
                            keyfile=KEYFILE,
                            certfile=CERTFILE,
                            server_side=True
                            )

    # Wait for connections
    while True:
        try:
            c,a = s_ssl.accept()
            print('Got connection', c, a)
            echo_client(c)
        except Exception as e:
            print('{}: {}'.format(e.__class__.__name__, e))

echo_server(('', 20000))
```

Here's an interactive session that shows how to connect to the server as a client. The client requires the server to present its certificate and verifies it:

```
>>> from socket import socket, AF_INET, SOCK_STREAM
>>> import ssl
>>> s = socket(AF_INET, SOCK_STREAM)
>>> s_ssl = ssl.wrap_socket(s,
...                          cert_reqs=ssl.CERT_REQUIRED,
...                          ca_certs = 'server_cert.pem')
>>> s_ssl.connect(('localhost', 20000))
>>> s_ssl.send(b'Hello World?')
12
>>> s_ssl.recv(8192)
b'Hello World?'
>>>
```

The problem with all of this low-level socket hacking is that it doesn't play well with existing network services already implemented in the standard library. For example, most server code (HTTP, XML-RPC, etc.) is actually based on the `socketserver` library. Client code is also implemented at a higher level. It is possible to add SSL to existing services, but a slightly different approach is needed.

First, for servers, SSL can be added through the use of a mixin class like this:

```
import ssl

class SSLMixin:
    """
    Mixin class that adds support for SSL to existing servers based
    on the socketserver module.
    """
    def __init__(self, *args,
                 keyfile=None, certfile=None, ca_certs=None,
                 cert_reqs=ssl.NONE,
                 **kwargs):
        self._keyfile = keyfile
        self._certfile = certfile
        self._ca_certs = ca_certs
        self._cert_reqs = cert_reqs
        super().__init__(*args, **kwargs)

    def get_request(self):
        client, addr = super().get_request()
        client_ssl = ssl.wrap_socket(client,
                                     keyfile = self._keyfile,
                                     certfile = self._certfile,
                                     ca_certs = self._ca_certs,
                                     cert_reqs = self._cert_reqs,
                                     server_side = True)

        return client_ssl, addr
```

To use this mixin class, you can mix it with other server classes. For example, here's an example of defining an XML-RPC server that operates over SSL:

```
# XML-RPC server with SSL

from xmlrpc.server import SimpleXMLRPCServer

class SSLSimpleXMLRPCServer(SSLMixin, SimpleXMLRPCServer):
    pass
```

Here's the XML-RPC server from [Recipe 11.6](#) modified only slightly to use SSL:

```
import ssl
from xmlrpc.server import SimpleXMLRPCServer
from sslmixin import SSLMixin

class SSLSimpleXMLRPCServer(SSLMixin, SimpleXMLRPCServer):
    pass

class KeyValueServer:
    _rpc_methods_ = ['get', 'set', 'delete', 'exists', 'keys']
    def __init__(self, *args, **kwargs):
        self._data = {}
        self._serv = SSLSimpleXMLRPCServer(*args, allow_none=True, **kwargs)
        for name in self._rpc_methods_:
            self._serv.register_function(getattr(self, name))

    def get(self, name):
        return self._data[name]

    def set(self, name, value):
        self._data[name] = value

    def delete(self, name):
        del self._data[name]

    def exists(self, name):
        return name in self._data

    def keys(self):
        return list(self._data)

    def serve_forever(self):
        self._serv.serve_forever()

if __name__ == '__main__':
    KEYFILE='server_key.pem' # Private key of the server
    CERTFILE='server_cert.pem' # Server certificate
    kvserv = KeyValueServer(('', 15000),
                            keyfile=KEYFILE,
                            certfile=CERTFILE),
    kvserv.serve_forever()
```

To use this server, you can connect using the normal `xmlrpc.client` module. Just specify a `https:` in the URL. For example:

```
>>> from xmlrpc.client import ServerProxy
>>> s = ServerProxy('https://localhost:15000', allow_none=True)
>>> s.set('foo', 'bar')
>>> s.set('spam', [1, 2, 3])
>>> s.keys()
['spam', 'foo']
>>> s.get('foo')
'bar'
>>> s.get('spam')
[1, 2, 3]
>>> s.delete('spam')
>>> s.exists('spam')
False
>>>
```

One complicated issue with SSL clients is performing extra steps to verify the server certificate or to present a server with client credentials (such as a client certificate). Unfortunately, there seems to be no standardized way to accomplish this, so research is often required. However, here is an example of how to set up a secure XML-RPC connection that verifies the server's certificate:

```
from xmlrpc.client import SafeTransport, ServerProxy
import ssl

class VerifyCertSafeTransport(SafeTransport):
    def __init__(self, cafile, certfile=None, keyfile=None):
        SafeTransport.__init__(self)
        self._ssl_context = ssl.SSLContext(ssl.PROTOCOL_TLSv1)
        self._ssl_context.load_verify_locations(cafile)
        if cert:
            self._ssl_context.load_cert_chain(certfile, keyfile)
        self._ssl_context.verify_mode = ssl.CERT_REQUIRED

    def make_connection(self, host):
        # Items in the passed dictionary are passed as keyword
        # arguments to the http.client.HTTPSConnection() constructor.
        # The context argument allows an ssl.SSLContext instance to
        # be passed with information about the SSL configuration
        s = super().make_connection((host, {'context': self._ssl_context}))

        return s

# Create the client proxy
s = ServerProxy('https://localhost:15000',
                transport=VerifyCertSafeTransport('server_cert.pem'),
                allow_none=True)
```

As shown, the server presents a certificate to the client and the client verifies it. This verification can go both directions. If the server wants to verify the client, change the server startup to the following:

```
if __name__ == '__main__':
    KEYFILE='server_key.pem' # Private key of the server
    CERTFILE='server_cert.pem' # Server certificate
    CA_CERTS='client_cert.pem' # Certificates of accepted clients

    kvserv = KeyValueServer((' ', 15000),
                           keyfile=KEYFILE,
                           certfile=CERTFILE,
                           ca_certs=CA_CERTS,
                           cert_reqs=ssl.CERT_REQUIRED,
                           )
    kvserv.serve_forever()
```

To make the XML-RPC client present its certificates, change the ServerProxy initialization to this:

```
# Create the client proxy
s = ServerProxy('https://localhost:15000',
               transport=VerifyCertSafeTransport('server_cert.pem',
                                                  'client_cert.pem',
                                                  'client_key.pem'),
               allow_none=True)
```

## Discussion

Getting this recipe to work will test your system configuration skills and understanding of SSL. Perhaps the biggest challenge is simply getting the initial configuration of keys, certificates, and other matters in order.

To clarify what's required, each endpoint of an SSL connection typically has a private key and a signed certificate file. The certificate file contains the public key and is presented to the remote peer on each connection. For public servers, certificates are normally signed by a certificate authority such as Verisign, Equifax, or similar organization (something that costs money). To verify server certificates, clients maintain a file containing the certificates of trusted certificate authorities. For example, web browsers maintain certificates corresponding to the major certificate authorities and use them to verify the integrity of certificates presented by web servers during HTTPS connections.

For the purposes of this recipe, you can create what's known as a self-signed certificate. Here's how you do it:

```
bash % openssl req -new -x509 -days 365 -nodes -out server_cert.pem \
      -keyout server_key.pem
Generating a 1024 bit RSA private key
.....++++++
...++++++
```

```
writing new private key to 'server_key.pem'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:Illinois
Locality Name (eg, city) []:Chicago
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Dabeaz, LLC
Organizational Unit Name (eg, section) []:
Common Name (eg, YOUR name) []:localhost
Email Address []:
bash %
```

When creating the certificate, the values for the various fields are often arbitrary. However, the “Common Name” field often contains the DNS hostname of servers. If you’re just testing things out on your own machine, use “localhost.” Otherwise, use the domain name of the machine that’s going to run the server.

As a result of this configuration, you will have a *server\_key.pem* file that contains the private key. It looks like this:

```
-----BEGIN RSA PRIVATE KEY-----
MIICXQIBAAKBgQCZrCNLoEyAKF+f9UNcFaz50sa6jf7qkbUl8si5xQrY3ZYC7juu
nL1dZLn/VbEFIIItAU0gvBTpV1qUWTJGwga62VSG1oFE00DIx3g2Nh4sRf+rySsx2
L4442nx0z405vJQ7k6ERNHAZUUnCL50+YvjyLyt7ryLSjSuKhCcJsbZgPwIDAQAB
AoGAB5evrr7eyL4160tM5rHteATlaLY3UB0e5Z8XN8Z6gLiB/ucSX9AysviVD/6F
3oD6z2al8jbeJc1vHqjt0dC2dwwm32vVl8mRdyoAsQpWmiqXrkvP4BsL04VpBeHw
Qt8xNSW9SFhcelL3LEvw9M8i9MV39viih1ILyH8OuHdvJyFECQQDLejl2d2ppxND9
PoLqVFAirDfX2JnLTdWbc+M11a9Jdn3hKF8TcxFeNFVs5Gav1MusicY5KB0ylYPb
YbTvqKc7AkeAwbnRB02VYEZsJZp2X0IZqP9ovWokkpYx+PE4+c6MySDgaMcigL7v
WDIHJG1CHudD09GbqENasDzyb2HAIW4CzQJBAKDDkv+Xow6Gjx42Auc2WzTcUHCA
eXR/+BLpPrhKykbzvQ8YvS5W764SU01u1LWs3G+wnRMvrRvLMCZKgggBjkCQQCG
Jewto2+a+Wk0KQXrNNScCDE5aPTmZQc5waCYq4UmCZQc0jku0iN3ST1U5iuxRqfb
V/yX6fw0qh+fLWtkOs/JAKA+okMSxZwqRtfg0FGBfwQ8/iKrnizeantQ3L6scFXI
CHZXDJ3XQ6qUmNnN7iJ7S/LDawo1QfWkCfD9FYoxBlg
-----END RSA PRIVATE KEY-----
```

The server certificate in *server\_cert.pem* looks similar:

```
-----BEGIN CERTIFICATE-----
MIIC+DCCAmGgAwIBAgIJAPMd+vi45js3MA0GCSqGSIb3DQEBBQUAMFwxCzAJBgNV
BAYTAlVTMRERDwYDVQQIEWhJbGxpbnM9pczEQMA4GA1UEBxMHQ2hpY2FnbzEUMBIG
A1UEChMLRGFiZWZ6L6CBMTEMxEjAQBgnVBAMTCWxvY2FsaG9zdAeFw0xMzAxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0x
NDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQx
MTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEw
ODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQy
MjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjda
Fw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw0xNDQxMTEwODQyMjdaFw
```

```
050nkTRwGVFJwi+dPmL48i8re68i0o0rioQnCbG2YD8CAwEAAa0BwTCBvjAdBgNV
Hq4EFgQURtoLHHgXiDZTr26NMmgKJLJLfIwgY4GA1UdIwSBhjCBg4AURtoLHHgX
iDZTr26NMmgKJLJLfKtKHYKReMFwxCzAJBgNVBAYTALVMTREwDwYDQVQIEWhJbGxp
bm9pczEQMA4GA1UEBxMHQ2hpY2FnbzEUMBIGA1UEChMLRGFiZWw6LCBMTEMxEjAQ
BgNVBAMTCWxvY2FsaG9zdIIJAPMd+vi45js3MAwGA1UdEwQFMAMBAF8wDQYJKoZI
hvcNAQEFBQADgYEAFCi+dqvMG4xF8UTnbGVvZJPIzJDRee6Nbt6AHQo9p0DAIMau
WsGCplSOaDNdKKzl+b2UT2Zp3AIW4Qd51bouSNnR4M/gnr9ZD1ZctFd3js+C5XRp
D3vvcw5LAnCCC80P6rXy7d7hTeFu5EYKtRGXNvVNd/06NALGDFlrr0wxF3Y=
-----END CERTIFICATE-----
```

In server-related code, both the private key and certificate file will be presented to the various SSL-related wrapping functions. The certificate is what gets presented to clients. The private key should be protected and remains on the server.

In client-related code, a special file of valid certificate authorities needs to be maintained to verify the server's certificate. If you have no such file, then at the very least, you can put a copy of the server's certificate on the client machine and use that as a means for verification. During connection, the server will present its certificate, and then you'll use the stored certificate you already have to verify that it's correct.

Servers can also elect to verify the identity of clients. To do that, clients need to have their own private key and certificate key. The server would also need to maintain a file of trusted certificate authorities for verifying the client certificates.

If you intend to add SSL support to a network service for real, this recipe really only gives a small taste of how to set it up. You will definitely want to consult [the documentation](#) for more of the finer points. Be prepared to spend a significant amount of time experimenting with it to get things to work.

## 11.11. Passing a Socket File Descriptor Between Processes

### Problem

You have multiple Python interpreter processes running and you want to pass an open file descriptor from one interpreter to the other. For instance, perhaps there is a server process that is responsible for receiving connections, but the actual servicing of clients is to be handled by a different interpreter.

### Solution

To pass a file descriptor between processes, you first need to connect the processes together. On Unix machines, you might use a Unix domain socket, whereas on Windows, you could use a named pipe. However, rather than deal with such low-level mechanics, it is often easier to use the `multiprocessing` module to set up such a connection.



Once a connection is established, you can use the `send_handle()` and `recv_handle()` functions in `multiprocessing.reduction` to send file descriptors between processes.

The following example illustrates the basics:

```
import multiprocessing
from multiprocessing.reduction import recv_handle, send_handle
import socket

def worker(in_p, out_p):
    out_p.close()
    while True:
        fd = recv_handle(in_p)
        print('CHILD: GOT FD', fd)
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM, fileno=fd) as s:
            while True:
                msg = s.recv(1024)
                if not msg:
                    break
                print('CHILD: RECV {!r}'.format(msg))
                s.send(msg)

def server(address, in_p, out_p, worker_pid):
    in_p.close()
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
    s.bind(address)
    s.listen(1)
    while True:
        client, addr = s.accept()
        print('SERVER: Got connection from', addr)
        send_handle(out_p, client.fileno(), worker_pid)
        client.close()

if __name__ == '__main__':
    c1, c2 = multiprocessing.Pipe()
    worker_p = multiprocessing.Process(target=worker, args=(c1, c2))
    worker_p.start()

    server_p = multiprocessing.Process(target=server,
                                       args=('', 15000), c1, c2, worker_p.pid)
    server_p.start()

    c1.close()
    c2.close()
```

In this example, two processes are spawned and connected by a `multiprocessing.Pipe` object. The server process opens a socket and waits for client connections. The worker process merely waits to receive a file descriptor on the pipe using `recv_handle()`. When the server receives a connection, it sends the resulting socket file descriptor to the worker

using `send_handle()`. The worker takes over the socket and echoes data back to the client until the connection is closed.

If you connect to the running server using Telnet or a similar tool, here is an example of what you might see:

```
bash % python3 passfd.py
SERVER: Got connection from ('127.0.0.1', 55543)
CHILD: GOT FD 7
CHILD: RECV b'Hello\r\n'
CHILD: RECV b'World\r\n'
```

The most important part of this example is the fact that the client socket accepted in the server is actually serviced by a completely different process. The server merely hands it off, closes it, and waits for the next connection.

## Discussion

Passing file descriptors between processes is something that many programmers don't even realize is possible. However, it can sometimes be a useful tool in building scalable systems. For example, on a multicore machine, you could have multiple instances of the Python interpreter and use file descriptor passing to more evenly balance the number of clients being handled by each interpreter.

The `send_handle()` and `recv_handle()` functions shown in the solution really only work with multiprocessing connections. Instead of using a pipe, you can connect interpreters as shown in [Recipe 11.7](#), and it will work as long as you use UNIX domain sockets or Windows pipes. For example, you could implement the server and worker as completely separate programs to be started separately. Here is the implementation of the server:

```
# servermp.py
from multiprocessing.connection import Listener
from multiprocessing.reduction import send_handle
import socket

def server(work_address, port):
    # Wait for the worker to connect
    work_serv = Listener(work_address, authkey=b'peekaboo')
    worker = work_serv.accept()
    worker_pid = worker.recv()

    # Now run a TCP/IP server and send clients to worker
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
    s.bind(('', port))
    s.listen(1)
    while True:
        client, addr = s.accept()
        print('SERVER: Got connection from', addr)
```

```

        send_handle(worker, client.fileno(), worker_pid)
        client.close()

if __name__ == '__main__':
    import sys
    if len(sys.argv) != 3:
        print('Usage: server.py server_address port', file=sys.stderr)
        raise SystemExit(1)

    server(sys.argv[1], int(sys.argv[2]))

```

To run this server, you would run a command such as `python3 servermp.py /tmp/servconn 15000`. Here is the corresponding client code:

```

# workermp.py

from multiprocessing.connection import Client
from multiprocessing.reduction import recv_handle
import os
from socket import socket, AF_INET, SOCK_STREAM

def worker(server_address):
    serv = Client(server_address, authkey=b'peekaboo')
    serv.send(os.getpid())
    while True:
        fd = recv_handle(serv)
        print('WORKER: GOT FD', fd)
        with socket(AF_INET, SOCK_STREAM, fileno=fd) as client:
            while True:
                msg = client.recv(1024)
                if not msg:
                    break
                print('WORKER: RECV {!r}'.format(msg))
                client.send(msg)

if __name__ == '__main__':
    import sys
    if len(sys.argv) != 2:
        print('Usage: worker.py server_address', file=sys.stderr)
        raise SystemExit(1)

    worker(sys.argv[1])

```

To run the worker, you would type `python3 workermp.py /tmp/servconn`. The resulting operation should be exactly the same as the example that used `Pipe()`.

Under the covers, file descriptor passing involves creating a UNIX domain socket and the `sendmsg()` method of sockets. Since this technique is not widely known, here is a different implementation of the server that shows how to pass descriptors using sockets:

```

# server.py
import socket

```

```

import struct

def send_fd(sock, fd):
    """
    Send a single file descriptor.
    """
    sock.sendmsg([b'x'],
                  [(socket.SOL_SOCKET, socket.SCM_RIGHTS, struct.pack('i', fd))])
    ack = sock.recv(2)
    assert ack == b'OK'

def server(work_address, port):
    # Wait for the worker to connect
    work_serv = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
    work_serv.bind(work_address)
    work_serv.listen(1)
    worker, addr = work_serv.accept()

    # Now run a TCP/IP server and send clients to worker
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
    s.bind(('', port))
    s.listen(1)
    while True:
        client, addr = s.accept()
        print('SERVER: Got connection from', addr)
        send_fd(worker, client.fileno())
        client.close()

if __name__ == '__main__':
    import sys
    if len(sys.argv) != 3:
        print('Usage: server.py server_address port', file=sys.stderr)
        raise SystemExit(1)

    server(sys.argv[1], int(sys.argv[2]))

```

Here is an implementation of the worker using sockets:

```

# worker.py
import socket
import struct

def recv_fd(sock):
    """
    Receive a single file descriptor
    """
    msg, ancdata, flags, addr = sock.recvmsg(1,
                                              socket.CMSG_LEN(struct.calcsize('i')))

    cmsg_level, cmsg_type, cmsg_data = ancdata[0]
    assert cmsg_level == socket.SOL_SOCKET and cmsg_type == socket.SCM_RIGHTS
    sock.sendall(b'OK')

```

```

        return struct.unpack('i', cmsg_data)[0]

def worker(server_address):
    serv = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
    serv.connect(server_address)
    while True:
        fd = recv_fd(serv)
        print('WORKER: GOT FD', fd)
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM, fileno=fd) as client:
            while True:
                msg = client.recv(1024)
                if not msg:
                    break
                print('WORKER: RECV {!r}'.format(msg))
                client.send(msg)

if __name__ == '__main__':
    import sys
    if len(sys.argv) != 2:
        print('Usage: worker.py server_address', file=sys.stderr)
        raise SystemExit(1)

    worker(sys.argv[1])

```

If you are going to use file-descriptor passing in your program, it is advisable to read more about it in an advanced text, such as *Unix Network Programming* by W. Richard Stevens (Prentice Hall, 1990). Passing file descriptors on Windows uses a different technique than Unix (not shown). For that platform, it is advisable to study the source code to `multiprocessing.reduction` in close detail to see how it works.

## 11.12. Understanding Event-Driven I/O

### Problem

You have heard about packages based on “event-driven” or “asynchronous” I/O, but you’re not entirely sure what it means, how it actually works under the covers, or how it might impact your program if you use it.

### Solution

At a fundamental level, event-driven I/O is a technique that takes basic I/O operations (e.g., reads and writes) and converts them into events that must be handled by your program. For example, whenever data was received on a socket, it turns into a “receive” event that is handled by some sort of callback method or function that you supply to respond to it. As a possible starting point, an event-driven framework might start with a base class that implements a series of basic event handler methods like this:

```

class EventHandler:
    def fileno(self):
        'Return the associated file descriptor'
        raise NotImplemented('must implement')

    def wants_to_receive(self):
        'Return True if receiving is allowed'
        return False

    def handle_receive(self):
        'Perform the receive operation'
        pass

    def wants_to_send(self):
        'Return True if sending is requested'
        return False

    def handle_send(self):
        'Send outgoing data'
        pass

```

Instances of this class then get plugged into an event loop that looks like this:

```

import select

def event_loop(handlers):
    while True:
        wants_recv = [h for h in handlers if h.wants_to_receive()]
        wants_send = [h for h in handlers if h.wants_to_send()]
        can_recv, can_send, _ = select.select(wants_recv, wants_send, [])
        for h in can_recv:
            h.handle_receive()
        for h in can_send:
            h.handle_send()

```

That's it! The key to the event loop is the `select()` call, which polls file descriptors for activity. Prior to calling `select()`, the event loop simply queries all of the handlers to see which ones want to receive or send. It then supplies the resulting lists to `select()`. As a result, `select()` returns the list of objects that are ready to receive or send. The corresponding `handle_receive()` or `handle_send()` methods are triggered.

To write applications, specific instances of `EventHandler` classes are created. For example, here are two simple handlers that illustrate two UDP-based network services:

```

import socket
import time

class UDPServer(EventHandler):
    def __init__(self, address):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.sock.bind(address)

```

```

def fileno(self):
    return self.sock.fileno()

def wants_to_receive(self):
    return True

class UDPTimeServer(UDPServer):
    def handle_receive(self):
        msg, addr = self.sock.recvfrom(1)
        self.sock.sendto(time.ctime().encode('ascii'), addr)

class UDPEchoServer(UDPServer):
    def handle_receive(self):
        msg, addr = self.sock.recvfrom(8192)
        self.sock.sendto(msg, addr)

if __name__ == '__main__':
    handlers = [ UDPTimeServer((' ', 14000)), UDPEchoServer((' ', 15000)) ]
    event_loop(handlers)

```

To test this code, you can try connecting to it from another Python interpreter:

```

>>> from socket import *
>>> s = socket(AF_INET, SOCK_DGRAM)
>>> s.sendto(b'', ('localhost', 14000))
0
>>> s.recvfrom(128)
(b'Tue Sep 18 14:29:23 2012', ('127.0.0.1', 14000))
>>> s.sendto(b'Hello', ('localhost', 15000))
5
>>> s.recvfrom(128)
(b'Hello', ('127.0.0.1', 15000))
>>>

```

Implementing a TCP server is somewhat more complex, since each client involves the instantiation of a new handler object. Here is an example of a TCP echo client.

```

class TCPServer(EventHandler):
    def __init__(self, address, client_handler, handler_list):
        self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, True)
        self.sock.bind(address)
        self.sock.listen(1)
        self.client_handler = client_handler
        self.handler_list = handler_list

    def fileno(self):
        return self.sock.fileno()

    def wants_to_receive(self):
        return True

```

```

def handle_receive(self):
    client, addr = self.sock.accept()
    # Add the client to the event loop's handler list
    self.handler_list.append(self.client_handler(client, self.handler_list))

class TCPClient(EventHandler):
    def __init__(self, sock, handler_list):
        self.sock = sock
        self.handler_list = handler_list
        self.outgoing = bytearray()

    def fileno(self):
        return self.sock.fileno()

    def close(self):
        self.sock.close()
        # Remove myself from the event loop's handler list
        self.handler_list.remove(self)

    def wants_to_send(self):
        return True if self.outgoing else False

    def handle_send(self):
        nsent = self.sock.send(self.outgoing)
        self.outgoing = self.outgoing[nsent:]

class TCPEchoClient(TCPClient):
    def wants_to_receive(self):
        return True

    def handle_receive(self):
        data = self.sock.recv(8192)
        if not data:
            self.close()
        else:
            self.outgoing.extend(data)

if __name__ == '__main__':
    handlers = []
    handlers.append(TCPServer(('', 16000), TCPEchoClient, handlers))
    event_loop(handlers)

```

The key to the TCP example is the addition and removal of clients from the handler list. On each connection, a new handler is created for the client and added to the list. When the connection is closed, each client must take care to remove themselves from the list.

If you run this program and try connecting with Telnet or some similar tool, you'll see it echoing received data back to you. It should easily handle multiple clients.



## Discussion

Virtually all event-driven frameworks operate in a manner that is similar to that shown in the solution. The actual implementation details and overall software architecture might vary greatly, but at the core, there is a polling loop that checks sockets for activity and which performs operations in response.

One potential benefit of event-driven I/O is that it can handle a very large number of simultaneous connections without ever using threads or processes. That is, the `select()` call (or equivalent) can be used to monitor hundreds or thousands of sockets and respond to events occurring on any of them. Events are handled one at a time by the event loop, without the need for any other concurrency primitives.

The downside to event-driven I/O is that there is no true concurrency involved. If any of the event handler methods blocks or performs a long-running calculation, it blocks the progress of everything. There is also the problem of calling out to library functions that aren't written in an event-driven style. There is always the risk that some library call will block, causing the event loop to stall.

Problems with blocking or long-running calculations can be solved by sending the work out to a separate thread or process. However, coordinating threads and processes with an event loop is tricky. Here is an example of code that will do it using the `concurrent.futures` module:

```
from concurrent.futures import ThreadPoolExecutor
import os

class ThreadPoolHandler(EventHandler):
    def __init__(self, nworkers):
        if os.name == 'posix':
            self.signal_done_sock, self.done_sock = socket.socketpair()
        else:
            server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            server.bind(('127.0.0.1', 0))
            server.listen(1)
            self.signal_done_sock = socket.socket(socket.AF_INET,
                                                  socket.SOCK_STREAM)
            self.signal_done_sock.connect(server.getsockname())
            self.done_sock, _ = server.accept()
            server.close()

        self.pending = []
        self.pool = ThreadPoolExecutor(nworkers)

    def fileno(self):
        return self.done_sock.fileno()

    # Callback that executes when the thread is done
    def _complete(self, callback, r):
```

```

self.pending.append((callback, r.result()))
self.signal_done_sock.send(b'x')

# Run a function in a thread pool
def run(self, func, args=(), kwargs={}, *, callback):
    r = self.pool.submit(func, *args, **kwargs)
    r.add_done_callback(lambda r: self._complete(callback, r))

def wants_to_receive(self):
    return True

# Run callback functions of completed work
def handle_receive(self):
    # Invoke all pending callback functions
    for callback, result in self.pending:
        callback(result)
        self.done_sock.recv(1)
    self.pending = []

```

In this code, the `run()` method is used to submit work to the pool along with a callback function that should be triggered upon completion. The actual work is then submitted to a `ThreadPoolExecutor` instance. However, a really tricky problem concerns the coordination of the computed result and the event loop. To do this, a pair of sockets are created under the covers and used as a kind of signaling mechanism. When work is completed by the thread pool, it executes the `_complete()` method in the class. This method queues up the pending callback and result before writing a byte of data on one of these sockets. The `fileno()` method is programmed to return the other socket. Thus, when this byte is written, it will signal to the event loop that something has happened. The `handle_receive()` method, when triggered, will then execute all of the callback functions for previously submitted work. Frankly, it's enough to make one's head spin.

Here is a simple server that shows how to use the thread pool to carry out a long-running calculation:

```

# A really bad Fibonacci implementation
def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)

class UDPFibServer(UDPServer):
    def handle_receive(self):
        msg, addr = self.sock.recvfrom(128)
        n = int(msg)
        pool.run(fib, (n,), callback=lambda r: self.respond(r, addr))

    def respond(self, result, addr):
        self.sock.sendto(str(result).encode('ascii'), addr)

```

```

if __name__ == '__main__':
    pool = ThreadPoolHandler(16)
    handlers = [ pool, UDPFibServer('', 16000)]
    event_loop(handlers)

```

To try this server, simply run it and try some experiments with another Python program:

```

from socket import *
sock = socket(AF_INET, SOCK_DGRAM)
for x in range(40):
    sock.sendto(str(x).encode('ascii'), ('localhost', 16000))
    resp = sock.recvfrom(8192)
    print(resp[0])

```

You should be able to run this program repeatedly from many different windows and have it operate without stalling other programs, even though it gets slower and slower as the numbers get larger.

Having gone through this recipe, should you use its code? Probably not. Instead, you should look for a more fully developed framework that accomplishes the same task. However, if you understand the basic concepts presented here, you'll understand the core techniques used to make such frameworks operate. As an alternative to callback-based programming, event-driven code will sometimes use coroutines. See [Recipe 12.12](#) for an example.

## 11.13. Sending and Receiving Large Arrays

### Problem

You want to send and receive large arrays of contiguous data across a network connection, making as few copies of the data as possible.

### Solution

The following functions utilize memoryviews to send and receive large arrays:

```

# zerocopy.py

def send_from(arr, dest):
    view = memoryview(arr).cast('B')
    while len(view):
        nsent = dest.send(view)
        view = view[nsent:]

def recv_into(arr, source):
    view = memoryview(arr).cast('B')
    while len(view):
        nrecv = source.recv_into(view)
        view = view[nrecv:]

```

To test the program, first create a server and client program connected over a socket. In the server:

```
>>> from socket import *
>>> s = socket(AF_INET, SOCK_STREAM)
>>> s.bind(('', 25000))
>>> s.listen(1)
>>> c,a = s.accept()
>>>
```

In the client (in a separate interpreter):

```
>>> from socket import *
>>> c = socket(AF_INET, SOCK_STREAM)
>>> c.connect(('localhost', 25000))
>>>
```

Now, the whole idea of this recipe is that you can blast a huge array through the connection. In this case, arrays might be created by the array module or perhaps numpy. For example:

```
# Server
>>> import numpy
>>> a = numpy.arange(0.0, 50000000.0)
>>> send_from(a, c)
>>>

# Client
>>> import numpy
>>> a = numpy.zeros(shape=50000000, dtype=float)
>>> a[0:10]
array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
>>> recv_into(a, c)
>>> a[0:10]
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
>>>
```

## Discussion

In data-intensive distributed computing and parallel programming applications, it's not uncommon to write programs that need to send/receive large chunks of data. However, to do this, you somehow need to reduce the data down to raw bytes for use with low-level network functions. You may also need to slice the data into chunks, since most network-related functions aren't able to send or receive huge blocks of data entirely all at once.

One approach is to serialize the data in some way—possibly by converting into a byte string. However, this usually ends up making a copy of the data. Even if you do this piecemeal, your code still ends up making a lot of little copies.

This recipe gets around this by playing a sneaky trick with memoryviews. Essentially, a memoryview is an overlay of an existing array. Not only that, memoryviews can be cast to different types to allow interpretation of the data in a different manner. This is the purpose of the following statement:

```
view = memoryview(arr).cast('B')
```

It takes an array `arr` and casts into a memoryview of unsigned bytes.

In this form, the view can be passed to socket-related functions, such as `sock.send()` or `sock.recv_into()`. Under the covers, those methods are able to work directly with the memory region. For example, `sock.send()` sends data directly from memory without a copy. `sock.recv_into()` uses the memoryview as the input buffer for the receive operation.

The remaining complication is the fact that the socket functions may only work with partial data. In general, it will take many different `send()` and `recv_into()` calls to transmit the entire array. Not to worry. After each operation, the view is sliced by the number of sent or received bytes to produce a new view. The new view is also a memory overlay. Thus, no copies are made.

One issue here is that the receiver has to know in advance how much data will be sent so that it can either preallocate an array or verify that it can receive the data into an existing array. If this is a problem, the sender could always arrange to send the size first, followed by the array data.



Python has long supported different approaches to concurrent programming, including programming with threads, launching subprocesses, and various tricks involving generator functions. In this chapter, recipes related to various aspects of concurrent programming are presented, including common thread programming techniques and approaches for parallel processing.

As experienced programmers know, concurrent programming is fraught with potential peril. Thus, a major focus of this chapter is on recipes that tend to lead to more reliable and debuggable code.

## 12.1. Starting and Stopping Threads

### Problem

You want to create and destroy threads for concurrent execution of code.

### Solution

The `threading` library can be used to execute any Python callable in its own thread. To do this, you create a `Thread` instance and supply the callable that you wish to execute as a target. Here is a simple example:

```
# Code to execute in an independent thread
import time
def countdown(n):
    while n > 0:
        print('T-minus', n)
        n -= 1
        time.sleep(5)
```

```
# Create and launch a thread
from threading import Thread
t = Thread(target=countdown, args=(10,))
t.start()
```

When you create a thread instance, it doesn't start executing until you invoke its `start()` method (which invokes the target function with the arguments you supplied).

Threads are executed in their own system-level thread (e.g., a POSIX thread or Windows threads) that is fully managed by the host operating system. Once started, threads run independently until the target function returns. You can query a thread instance to see if it's still running:

```
if t.is_alive():
    print('Still running')
else:
    print('Completed')
```

You can also request to join with a thread, which waits for it to terminate:

```
t.join()
```

The interpreter remains running until all threads terminate. For long-running threads or background tasks that run forever, you should consider making the thread *daemonic*. For example:

```
t = Thread(target=countdown, args=(10,), daemon=True)
t.start()
```

Daemonic threads can't be joined. However, they are destroyed automatically when the main thread terminates.

Beyond the two operations shown, there aren't many other things you can do with threads. For example, there are no operations to terminate a thread, signal a thread, adjust its scheduling, or perform any other high-level operations. If you want these features, you need to build them yourself.

If you want to be able to terminate threads, the thread must be programmed to poll for exit at selected points. For example, you might put your thread in a class such as this:

```
class CountdownTask:
    def __init__(self):
        self._running = True

    def terminate(self):
        self._running = False

    def run(self, n):
        while self._running and n > 0:
            print('T-minus', n)
            n -= 1
            time.sleep(5)
```



```

c = CountdownTask()
t = Thread(target=c.run, args=(10,))
t.start()
...
c.terminate() # Signal termination
t.join()      # Wait for actual termination (if needed)

```

Polling for thread termination can be tricky to coordinate if threads perform blocking operations such as I/O. For example, a thread blocked indefinitely on an I/O operation may never return to check if it's been killed. To correctly deal with this case, you'll need to carefully program thread to utilize timeout loops. For example:

```

class IOTask:
    def terminate(self):
        self._running = False

    def run(self, sock):
        # sock is a socket
        sock.settimeout(5)      # Set timeout period
        while self._running:
            # Perform a blocking I/O operation w/ timeout
            try:
                data = sock.recv(8192)
                break
            except socket.timeout:
                continue
            # Continued processing
        ...
        # Terminated
        return

```

## Discussion

Due to a global interpreter lock (GIL), Python threads are restricted to an execution model that only allows one thread to execute in the interpreter at any given time. For this reason, Python threads should generally not be used for computationally intensive tasks where you are trying to achieve parallelism on multiple CPUs. They are much better suited for I/O handling and handling concurrent execution in code that performs blocking operations (e.g., waiting for I/O, waiting for results from a database, etc.).

Sometimes you will see threads defined via inheritance from the Thread class. For example:

```

from threading import Thread

class CountdownThread(Thread):
    def __init__(self, n):
        super().__init__()
        self.n = 0
    def run(self):
        while self.n > 0:

```

```

        print('T-minus', self.n)
        self.n -= 1
        time.sleep(5)

c = CountdownThread(5)
c.start()

```

Although this works, it introduces an extra dependency between the code and the `threading` library. That is, you can only use the resulting code in the context of threads, whereas the technique shown earlier involves writing code with no explicit dependency on `threading`. By freeing your code of such dependencies, it becomes usable in other contexts that may or may not involve threads. For instance, you might be able to execute your code in a separate process using the `multiprocessing` module using code like this:

```

import multiprocessing
c = CountdownTask(5)
p = multiprocessing.Process(target=c.run)
p.start()
...

```

Again, this only works if the `CountdownTask` class has been written in a manner that is neutral to the actual means of concurrency (threads, processes, etc.).

## 12.2. Determining If a Thread Has Started

### Problem

You’ve launched a thread, but want to know when it actually starts running.

### Solution

A key feature of threads is that they execute independently and nondeterministically. This can present a tricky synchronization problem if other threads in the program need to know if a thread has reached a certain point in its execution before carrying out further operations. To solve such problems, use the `Event` object from the `threading` library.

`Event` instances are similar to a “sticky” flag that allows threads to wait for something to happen. Initially, an event is set to 0. If the event is unset and a thread waits on the event, it will block (i.e., go to sleep) until the event gets set. A thread that sets the event will wake up all of the threads that happen to be waiting (if any). If a thread waits on an event that has already been set, it merely moves on, continuing to execute.

Here is some sample code that uses an `Event` to coordinate the startup of a thread:

```

from threading import Thread, Event
import time

```

```

# Code to execute in an independent thread
def countdown(n, started_evt):
    print('countdown starting')
    started_evt.set()
    while n > 0:
        print('T-minus', n)
        n -= 1
        time.sleep(5)

# Create the event object that will be used to signal startup
started_evt = Event()

# Launch the thread and pass the startup event
print('Launching countdown')
t = Thread(target=countdown, args=(10,started_evt))
t.start()

# Wait for the thread to start
started_evt.wait()
print('countdown is running')

```

When you run this code, the “countdown is running” message will always appear after the “countdown starting” message. This is coordinated by the event that makes the main thread wait until the `countdown()` function has first printed the startup message.

## Discussion

Event objects are best used for one-time events. That is, you create an event, threads wait for the event to be set, and once set, the `Event` is discarded. Although it is possible to clear an event using its `clear()` method, safely clearing an event and waiting for it to be set again is tricky to coordinate, and can lead to missed events, deadlock, or other problems (in particular, you can’t guarantee that a request to clear an event after setting it will execute before a released thread cycles back to wait on the event again).

If a thread is going to repeatedly signal an event over and over, you’re probably better off using a `Condition` object instead. For example, this code implements a periodic timer that other threads can monitor to see whenever the timer expires:

```

import threading
import time

class PeriodicTimer:
    def __init__(self, interval):
        self._interval = interval
        self._flag = 0
        self._cv = threading.Condition()

    def start(self):
        t = threading.Thread(target=self.run)
        t.daemon = True

```

```

        t.start()

    def run(self):
        """
        Run the timer and notify waiting threads after each interval
        """
        while True:
            time.sleep(self._interval)
            with self._cv:
                self._flag ^= 1
                self._cv.notify_all()

    def wait_for_tick(self):
        """
        Wait for the next tick of the timer
        """
        with self._cv:
            last_flag = self._flag
            while last_flag == self._flag:
                self._cv.wait()

# Example use of the timer
ptimer = PeriodicTimer(5)
ptimer.start()

# Two threads that synchronize on the timer
def countdown(nticks):
    while nticks > 0:
        ptimer.wait_for_tick()
        print('T-minus', nticks)
        nticks -= 1

def countup(last):
    n = 0
    while n < last:
        ptimer.wait_for_tick()
        print('Counting', n)
        n += 1

threading.Thread(target=countdown, args=(10,)).start()
threading.Thread(target=countup, args=(5,)).start()

```

A critical feature of Event objects is that they wake all waiting threads. If you are writing a program where you only want to wake up a single waiting thread, it is probably better to use a Semaphore or Condition object instead.

For example, consider this code involving semaphores:

```

# Worker thread
def worker(n, sema):
    # Wait to be signaled
    sema.acquire()

```

```

    # Do some work
    print('Working', n)

# Create some threads
    sema = threading.Semaphore(0)
    nworkers = 10
    for n in range(nworkers):
        t = threading.Thread(target=worker, args=(n, sema,))
        t.start()

```

If you run this, a pool of threads will start, but nothing happens because they're all blocked waiting to acquire the semaphore. Each time the semaphore is released, only one worker will wake up and run. For example:

```

>>> sema.release()
Working 0
>>> sema.release()
Working 1
>>>

```

Writing code that involves a lot of tricky synchronization between threads is likely to make your head explode. A more sane approach is to thread threads as communicating tasks using queues or as actors. Queues are described in the next recipe. Actors are described in [Recipe 12.10](#).

## 12.3. Communicating Between Threads

### Problem

You have multiple threads in your program and you want to safely communicate or exchange data between them.

### Solution

Perhaps the safest way to send data from one thread to another is to use a Queue from the queue library. To do this, you create a Queue instance that is shared by the threads. Threads then use `put()` or `get()` operations to add or remove items from the queue. For example:

```

from queue import Queue
from threading import Thread

# A thread that produces data
def producer(out_q):
    while True:
        # Produce some data
        ...
        out_q.put(data)

```

```

# A thread that consumes data
def consumer(in_q):
    while True:
        # Get some data
        data = in_q.get()
        # Process the data
        ...

# Create the shared queue and launch both threads
q = Queue()
t1 = Thread(target=consumer, args=(q,))
t2 = Thread(target=producer, args=(q,))
t1.start()
t2.start()

```

Queue instances already have all of the required locking, so they can be safely shared by as many threads as you wish.

When using queues, it can be somewhat tricky to coordinate the shutdown of the producer and consumer. A common solution to this problem is to rely on a special sentinel value, which when placed in the queue, causes consumers to terminate. For example:

```

from queue import Queue
from threading import Thread

# Object that signals shutdown
_sentinel = object()

# A thread that produces data
def producer(out_q):
    while running:
        # Produce some data
        ...
        out_q.put(data)

    # Put the sentinel on the queue to indicate completion
    out_q.put(_sentinel)

# A thread that consumes data
def consumer(in_q):
    while True:
        # Get some data
        data = in_q.get()

        # Check for termination
        if data is _sentinel:
            in_q.put(_sentinel)
            break

        # Process the data
        ...

```

A subtle feature of this example is that the consumer, upon receiving the special sentinel value, immediately places it back onto the queue. This propagates the sentinel to other consumers threads that might be listening on the same queue—thus shutting them all down one after the other.

Although queues are the most common thread communication mechanism, you can build your own data structures as long as you add the required locking and synchronization. The most common way to do this is to wrap your data structures with a condition variable. For example, here is how you might build a thread-safe priority queue, as discussed in [Recipe 1.5](#).

```
import heapq
import threading

class PriorityQueue:
    def __init__(self):
        self._queue = []
        self._count = 0
        self._cv = threading.Condition()
    def put(self, item, priority):
        with self._cv:
            heapq.heappush(self._queue, (-priority, self._count, item))
            self._count += 1
            self._cv.notify()

    def get(self):
        with self._cv:
            while len(self._queue) == 0:
                self._cv.wait()
            return heapq.heappop(self._queue)[-1]
```

Thread communication with a queue is a one-way and nondeterministic process. In general, there is no way to know when the receiving thread has actually received a message and worked on it. However, Queue objects do provide some basic completion features, as illustrated by the `task_done()` and `join()` methods in this example:

```
from queue import Queue
from threading import Thread

# A thread that produces data
def producer(out_q):
    while running:
        # Produce some data
        ...
        out_q.put(data)

# A thread that consumes data
def consumer(in_q):
    while True:
        # Get some data
        data = in_q.get()
```

```

        # Process the data
        ...
        # Indicate completion
        in_q.task_done()

# Create the shared queue and launch both threads
q = Queue()
t1 = Thread(target=consumer, args=(q,))
t2 = Thread(target=producer, args=(q,))
t1.start()
t2.start()

# Wait for all produced items to be consumed
q.join()

```

If a thread needs to know immediately when a consumer thread has processed a particular item of data, you should pair the sent data with an Event object that allows the producer to monitor its progress. For example:

```

from queue import Queue
from threading import Thread, Event

# A thread that produces data
def producer(out_q):
    while running:
        # Produce some data
        ...
        # Make an (data, event) pair and hand it to the consumer
        evt = Event()
        out_q.put((data, evt))
        ...
        # Wait for the consumer to process the item
        evt.wait()

# A thread that consumes data
def consumer(in_q):
    while True:
        # Get some data
        data, evt = in_q.get()
        # Process the data
        ...
        # Indicate completion
        evt.set()

```

## Discussion

Writing threaded programs based on simple queuing is often a good way to maintain sanity. If you can break everything down to simple thread-safe queuing, you'll find that you don't need to litter your program with locks and other low-level synchronization. Also, communicating with queues often leads to designs that can be scaled up to other kinds of message-based communication patterns later on. For instance, you might be



able to split your program into multiple processes, or even a distributed system, without changing much of its underlying queuing architecture.

One caution with thread queues is that putting an item in a queue doesn't make a copy of the item. Thus, communication actually involves passing an object reference between threads. If you are concerned about shared state, it may make sense to only pass immutable data structures (e.g., integers, strings, or tuples) or to make deep copies of the queued items. For example:

```
from queue import Queue
from threading import Thread
import copy

# A thread that produces data
def producer(out_q):
    while True:
        # Produce some data
        ...
        out_q.put(copy.deepcopy(data))

# A thread that consumes data
def consumer(in_q):
    while True:
        # Get some data
        data = in_q.get()
        # Process the data
        ...
```

Queue objects provide a few additional features that may prove to be useful in certain contexts. If you create a Queue with an optional size, such as Queue(N), it places a limit on the number of items that can be enqueued before the put() blocks the producer. Adding an upper bound to a queue might make sense if there is mismatch in speed between a producer and consumer. For instance, if a producer is generating items at a much faster rate than they can be consumed. On the other hand, making a queue block when it's full can also have an unintended cascading effect throughout your program, possibly causing it to deadlock or run poorly. In general, the problem of “flow control” between communicating threads is a much harder problem than it seems. If you ever find yourself trying to fix a problem by fiddling with queue sizes, it could be an indicator of a fragile design or some other inherent scaling problem.

Both the get() and put() methods support nonblocking and timeouts. For example:

```
import queue
q = queue.Queue()

try:
    data = q.get(block=False)
except queue.Empty:
    ...
```

```

try:
    q.put(item, block=False)
except queue.Full:
    ...

try:
    data = q.get(timeout=5.0)
except queue.Empty:
    ...

```

Both of these options can be used to avoid the problem of just blocking indefinitely on a particular queuing operation. For example, a nonblocking `put()` could be used with a fixed-sized queue to implement different kinds of handling code for when a queue is full. For example, issuing a log message and discarding:

```

def producer(q):
    ...
    try:
        q.put(item, block=False)
    except queue.Full:
        log.warning('queued item %r discarded!', item)

```

A timeout is useful if you're trying to make consumer threads periodically give up on operations such as `q.get()` so that they can check things such as a termination flag, as described in [Recipe 12.1](#).

```

_running = True

def consumer(q):
    while _running:
        try:
            item = q.get(timeout=5.0)
            # Process item
            ...
        except queue.Empty:
            pass

```

Lastly, there are utility methods `q.qsize()`, `q.full()`, `q.empty()` that can tell you the current size and status of the queue. However, be aware that all of these are unreliable in a multithreaded environment. For example, a call to `q.empty()` might tell you that the queue is empty, but in the time that has elapsed since making the call, another thread could have added an item to the queue. Frankly, it's best to write your code not to rely on such functions.

## 12.4. Locking Critical Sections

### Problem

Your program uses threads and you want to lock critical sections of code to avoid race conditions.

### Solution

To make mutable objects safe to use by multiple threads, use Lock objects in the threading library, as shown here:

```
import threading

class SharedCounter:
    """
    A counter object that can be shared by multiple threads.
    """
    def __init__(self, initial_value = 0):
        self._value = initial_value
        self._value_lock = threading.Lock()

    def incr(self, delta=1):
        """
        Increment the counter with locking
        """
        with self._value_lock:
            self._value += delta

    def decr(self, delta=1):
        """
        Decrement the counter with locking
        """
        with self._value_lock:
            self._value -= delta
```

A Lock guarantees mutual exclusion when used with the `with` statement—that is, only one thread is allowed to execute the block of statements under the `with` statement at a time. The `with` statement acquires the lock for the duration of the indented statements and releases the lock when control flow exits the indented block.

### Discussion

Thread scheduling is inherently nondeterministic. Because of this, failure to use locks in threaded programs can result in randomly corrupted data and bizarre behavior known as a “race condition.” To avoid this, locks should always be used whenever shared mutable state is accessed by multiple threads.

In older Python code, it is common to see locks explicitly acquired and released. For example, in this variant of the last example:

```
import threading

class SharedCounter:
    """
    A counter object that can be shared by multiple threads.
    """
    def __init__(self, initial_value = 0):
        self._value = initial_value
        self._value_lock = threading.Lock()

    def incr(self,delta=1):
        """
        Increment the counter with locking
        """
        self._value_lock.acquire()
        self._value += delta
        self._value_lock.release()

    def decr(self,delta=1):
        """
        Decrement the counter with locking
        """
        self._value_lock.acquire()
        self._value -= delta
        self._value_lock.release()
```

The `with` statement is more elegant and less prone to error—especially in situations where a programmer might forget to call the `release()` method or if a program happens to raise an exception while holding a lock (the `with` statement guarantees that locks are always released in both cases).

To avoid the potential for deadlock, programs that use locks should be written in a way such that each thread is only allowed to acquire one lock at a time. If this is not possible, you may need to introduce more advanced deadlock avoidance into your program, as described in [Recipe 12.5](#).

In the `threading` library, you’ll find other synchronization primitives, such as `RLock` and `Semaphore` objects. As a general rule of thumb, these are more special purpose and should not be used for simple locking of mutable state. An `RLock` or re-entrant lock object is a lock that can be acquired multiple times by the same thread. It is primarily used to implement code based locking or synchronization based on a construct known as a “monitor.” With this kind of locking, only one thread is allowed to use an entire function or the methods of a class while the lock is held. For example, you could implement the `SharedCounter` class like this:

```

import threading

class SharedCounter:
    """
    A counter object that can be shared by multiple threads.
    """
    _lock = threading.RLock()
    def __init__(self, initial_value = 0):
        self._value = initial_value

    def incr(self, delta=1):
        """
        Increment the counter with locking
        """
        with SharedCounter._lock:
            self._value += delta

    def decr(self, delta=1):
        """
        Decrement the counter with locking
        """
        with SharedCounter._lock:
            self.incr(-delta)

```

In this variant of the code, there is just a single class-level lock shared by all instances of the class. Instead of the lock being tied to the per-instance mutable state, the lock is meant to synchronize the methods of the class. Specifically, this lock ensures that only one thread is allowed to be using the methods of the class at once. However, unlike a standard lock, it is OK for methods that already have the lock to call other methods that also use the lock (e.g., see the `decr()` method).

One feature of this implementation is that only one lock is created, regardless of how many counter instances are created. Thus, it is much more memory-efficient in situations where there are a large number of counters. However, a possible downside is that it may cause more lock contention in programs that use a large number of threads and make frequent counter updates.

A Semaphore object is a synchronization primitive based on a shared counter. If the counter is nonzero, the `with` statement decrements the count and a thread is allowed to proceed. The counter is incremented upon the conclusion of the `with` block. If the counter is zero, progress is blocked until the counter is incremented by another thread. Although a semaphore can be used in the same manner as a standard Lock, the added complexity in implementation negatively impacts performance. Instead of simple locking, Semaphore objects are more useful for applications involving signaling between threads or throttling. For example, if you want to limit the amount of concurrency in a part of code, you might use a semaphore, as follows:

```

from threading import Semaphore
import urllib.request

```

```
# At most, five threads allowed to run at once
_fetch_url_sema = Semaphore(5)

def fetch_url(url):
    with _fetch_url_sema:
        return urllib.request.urlopen(url)
```

If you're interested in the underlying theory and implementation of thread synchronization primitives, consult almost any textbook on operating systems.

## 12.5. Locking with Deadlock Avoidance

### Problem

You're writing a multithreaded program where threads need to acquire more than one lock at a time while avoiding deadlock.

### Solution

In multithreaded programs, a common source of deadlock is due to threads that attempt to acquire multiple locks at once. For instance, if a thread acquires the first lock, but then blocks trying to acquire the second lock, that thread can potentially block the progress of other threads and make the program freeze.

One solution to deadlock avoidance is to assign each lock in the program a unique number, and to enforce an ordering rule that only allows multiple locks to be acquired in ascending order. This is surprisingly easy to implement using a context manager as follows:

```
import threading
from contextlib import contextmanager

# Thread-local state to stored information on locks already acquired
_local = threading.local()

@contextmanager
def acquire(*locks):
    # Sort locks by object identifier
    locks = sorted(locks, key=lambda x: id(x))

    # Make sure lock order of previously acquired locks is not violated
    acquired = getattr(_local, 'acquired', [])
    if acquired and max(id(lock) for lock in acquired) >= id(locks[0]):
        raise RuntimeError('Lock Order Violation')

    # Acquire all of the locks
    acquired.extend(locks)
    _local.acquired = acquired
```

```

try:
    for lock in locks:
        lock.acquire()
    yield
finally:
    # Release locks in reverse order of acquisition
    for lock in reversed(locks):
        lock.release()
    del acquired[-len(locks):]

```

To use this context manager, you simply allocate lock objects in the normal way, but use the `acquire()` function whenever you want to work with one or more locks. For example:

```

import threading
x_lock = threading.Lock()
y_lock = threading.Lock()

def thread_1():
    while True:
        with acquire(x_lock, y_lock):
            print('Thread-1')

def thread_2():
    while True:
        with acquire(y_lock, x_lock):
            print('Thread-2')

t1 = threading.Thread(target=thread_1)
t1.daemon = True
t1.start()

t2 = threading.Thread(target=thread_2)
t2.daemon = True
t2.start()

```

If you run this program, you'll find that it happily runs forever without deadlock—even though the acquisition of locks is specified in a different order in each function.

The key to this recipe lies in the first statement that sorts the locks according to object identifier. By sorting the locks, they always get acquired in a consistent order regardless of how the user might have provided them to `acquire()`.

The solution uses thread-local storage to solve a subtle problem with detecting potential deadlock if multiple `acquire()` operations are nested. For example, suppose you wrote the code like this:

```

import threading
x_lock = threading.Lock()
y_lock = threading.Lock()

def thread_1():

```

```

while True:
    with acquire(x_lock):
        with acquire(y_lock):
            print('Thread-1')

def thread_2():
    while True:
        with acquire(y_lock):
            with acquire(x_lock):
                print('Thread-2')

t1 = threading.Thread(target=thread_1)
t1.daemon = True
t1.start()

t2 = threading.Thread(target=thread_2)
t2.daemon = True
t2.start()

```

If you run this version of the program, one of the threads will crash with an exception such as this:

```

Exception in thread Thread-1:
Traceback (most recent call last):
  File "/usr/local/lib/python3.3/threading.py", line 639, in _bootstrap_inner
    self.run()
  File "/usr/local/lib/python3.3/threading.py", line 596, in run
    self._target(*self._args, **self._kwargs)
  File "deadlock.py", line 49, in thread_1
    with acquire(y_lock):
  File "/usr/local/lib/python3.3/contextlib.py", line 48, in __enter__
    return next(self.gen)
  File "deadlock.py", line 15, in acquire
    raise RuntimeError("Lock Order Violation")
RuntimeError: Lock Order Violation
>>>

```

This crash is caused by the fact that each thread remembers the locks it has already acquired. The `acquire()` function checks the list of previously acquired locks and enforces the ordering constraint that previously acquired locks must have an object ID that is less than the new locks being acquired.

## Discussion

The issue of deadlock is a well-known problem with programs involving threads (as well as a common subject in textbooks on operating systems). As a rule of thumb, as long as you can ensure that threads can hold only one lock at a time, your program will be deadlock free. However, once multiple locks are being acquired at the same time, all bets are off.



Detecting and recovering from deadlock is an extremely tricky problem with few elegant solutions. For example, a common deadlock detection and recovery scheme involves the use of a watchdog timer. As threads run, they periodically reset the timer, and as long as everything is running smoothly, all is well. However, if the program deadlocks, the watchdog timer will eventually expire. At that point, the program “recovers” by killing and then restarting itself.

Deadlock avoidance is a different strategy where locking operations are carried out in a manner that simply does not allow the program to enter a deadlocked state. The solution in which locks are always acquired in strict order of ascending object ID can be mathematically proven to avoid deadlock, although the proof is left as an exercise to the reader (the gist of it is that by acquiring locks in a purely increasing order, you can’t get cyclic locking dependencies, which are a necessary condition for deadlock to occur).

As a final example, a classic thread deadlock problem is the so-called “dining philosopher’s problem.” In this problem, five philosophers sit around a table on which there are five bowls of rice and five chopsticks. Each philosopher represents an independent thread and each chopstick represents a lock. In the problem, philosophers either sit and think or they eat rice. However, in order to eat rice, a philosopher needs two chopsticks. Unfortunately, if all of the philosophers reach over and grab the chopstick to their left, they’ll all just sit there with one stick and eventually starve to death. It’s a gruesome scene.

Using the solution, here is a simple deadlock free implementation of the dining philosopher’s problem:

```
import threading

# The philosopher thread
def philosopher(left, right):
    while True:
        with acquire(left, right):
            print(threading.currentThread(), 'eating')

# The chopsticks (represented by locks)
NSTICKS = 5
chopsticks = [threading.Lock() for n in range(NSTICKS)]

# Create all of the philosophers
for n in range(NSTICKS):
    t = threading.Thread(target=philosopher,
                        args=(chopsticks[n], chopsticks[(n+1) % NSTICKS]))
    t.start()
```

Last, but not least, it should be noted that in order to avoid deadlock, all locking operations must be carried out using our `acquire()` function. If some fragment of code decided to acquire a lock directly, then the deadlock avoidance algorithm wouldn’t work.

## 12.6. Storing Thread-Specific State

### Problem

You need to store state that's specific to the currently executing thread and not visible to other threads.

### Solution

Sometimes in multithreaded programs, you need to store data that is only specific to the currently executing thread. To do this, create a thread-local storage object using `threading.local()`. Attributes stored and read on this object are only visible to the executing thread and no others.

As an interesting practical example of using thread-local storage, consider the `LazyConnection` context-manager class that was first defined in [Recipe 8.3](#). Here is a slightly modified version that safely works with multiple threads:

```
from socket import socket, AF_INET, SOCK_STREAM
import threading

class LazyConnection:
    def __init__(self, address, family=AF_INET, type=SOCK_STREAM):
        self.address = address
        self.family = AF_INET
        self.type = SOCK_STREAM
        self.local = threading.local()

    def __enter__(self):
        if hasattr(self.local, 'sock'):
            raise RuntimeError('Already connected')
        self.local.sock = socket(self.family, self.type)
        self.local.sock.connect(self.address)
        return self.local.sock

    def __exit__(self, exc_ty, exc_val, tb):
        self.local.sock.close()
        del self.local.sock
```

In this code, carefully observe the use of the `self.local` attribute. It is initialized as an instance of `threading.local()`. The other methods then manipulate a socket that's stored as `self.local.sock`. This is enough to make it possible to safely use an instance of `LazyConnection` in multiple threads. For example:

```
from functools import partial
def test(conn):
    with conn as s:
        s.send(b'GET /index.html HTTP/1.0\r\n')
        s.send(b'Host: www.python.org\r\n')
```

```

s.send(b'\r\n')
resp = b''.join(iter(partial(s.recv, 8192), b''))

print('Got {} bytes'.format(len(resp)))

if __name__ == '__main__':
    conn = LazyConnection(('www.python.org', 80))

    t1 = threading.Thread(target=test, args=(conn,))
    t2 = threading.Thread(target=test, args=(conn,))
    t1.start()
    t2.start()
    t1.join()
    t2.join()

```

The reason it works is that each thread actually creates its own dedicated socket connection (stored as `self.local.sock`). Thus, when the different threads perform socket operations, they don't interfere with one another as they are being performed on different sockets.

## Discussion

Creating and manipulating thread-specific state is not a problem that often arises in most programs. However, when it does, it commonly involves situations where an object being used by multiple threads needs to manipulate some kind of dedicated system resource, such as a socket or file. You can't just have a single socket object shared by everyone because chaos would ensue if multiple threads ever started reading and writing on it at the same time. Thread-local storage fixes this by making such resources only visible in the thread where they're being used.

In this recipe, the use of `threading.local()` makes the `LazyConnection` class support one connection per thread, as opposed to one connection for the entire process. It's a subtle but interesting distinction.

Under the covers, an instance of `threading.local()` maintains a separate instance dictionary for each thread. All of the usual instance operations of getting, setting, and deleting values just manipulate the per-thread dictionary. The fact that each thread uses a separate dictionary is what provides the isolation of data.

## 12.7. Creating a Thread Pool

### Problem

You want to create a pool of worker threads for serving clients or performing other kinds of work.

## Solution

The `concurrent.futures` library has a `ThreadPoolExecutor` class that can be used for this purpose. Here is an example of a simple TCP server that uses a thread-pool to serve clients:

```
from socket import AF_INET, SOCK_STREAM, socket
from concurrent.futures import ThreadPoolExecutor

def echo_client(sock, client_addr):
    """
    Handle a client connection
    """
    print('Got connection from', client_addr)
    while True:
        msg = sock.recv(65536)
        if not msg:
            break
        sock.sendall(msg)
    print('Client closed connection')
    sock.close()

def echo_server(addr):
    pool = ThreadPoolExecutor(128)
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(addr)
    sock.listen(5)
    while True:
        client_sock, client_addr = sock.accept()
        pool.submit(echo_client, client_sock, client_addr)

echo_server(('', 15000))
```

If you want to manually create your own thread pool, it's usually easy enough to do it using a `Queue`. Here is a slightly different, but manual implementation of the same code:

```
from socket import socket, AF_INET, SOCK_STREAM
from threading import Thread
from queue import Queue

def echo_client(q):
    """
    Handle a client connection
    """
    sock, client_addr = q.get()
    print('Got connection from', client_addr)
    while True:
        msg = sock.recv(65536)
        if not msg:
            break
        sock.sendall(msg)
    print('Client closed connection')
```

```

sock.close()

def echo_server(addr, nworkers):
    # Launch the client workers
    q = Queue()
    for n in range(nworkers):
        t = Thread(target=echo_client, args=(q,))
        t.daemon = True
        t.start()

    # Run the server
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(addr)
    sock.listen(5)
    while True:
        client_sock, client_addr = sock.accept()
        q.put((client_sock, client_addr))

echo_server((' ', 15000), 128)

```

One advantage of using `ThreadPoolExecutor` over a manual implementation is that it makes it easier for the submitter to receive results from the called function. For example, you could write code like this:

```

from concurrent.futures import ThreadPoolExecutor
import urllib.request

def fetch_url(url):
    u = urllib.request.urlopen(url)
    data = u.read()
    return data

pool = ThreadPoolExecutor(10)
# Submit work to the pool
a = pool.submit(fetch_url, 'http://www.python.org')
b = pool.submit(fetch_url, 'http://www.pypy.org')

# Get the results back
x = a.result()
y = b.result()

```

The result objects in the example handle all of the blocking and coordination needed to get data back from the worker thread. Specifically, the operation `a.result()` blocks until the corresponding function has been executed by the pool and returned a value.

## Discussion

Generally, you should avoid writing programs that allow unlimited growth in the number of threads. For example, take a look at the following server:

```

from threading import Thread
from socket import socket, AF_INET, SOCK_STREAM

```

```

def echo_client(sock, client_addr):
    """
    Handle a client connection
    """
    print('Got connection from', client_addr)
    while True:
        msg = sock.recv(65536)
        if not msg:
            break
        sock.sendall(msg)
    print('Client closed connection')
    sock.close()

def echo_server(addr, nworkers):
    # Run the server
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(addr)
    sock.listen(5)
    while True:
        client_sock, client_addr = sock.accept()
        t = Thread(target=echo_client, args=(client_sock, client_addr))
        t.daemon = True
        t.start()

echo_server(' ', 15000)

```

Although this works, it doesn't prevent some asynchronous hipster from launching an attack on the server that makes it create so many threads that your program runs out of resources and crashes (thus further demonstrating the “evils” of using threads). By using a pre-initialized thread pool, you can carefully put an upper limit on the amount of supported concurrency.

You might be concerned with the effect of creating a large number of threads. However, modern systems should have no trouble creating pools of a few thousand threads. Moreover, having a thousand threads just sitting around waiting for work isn't going to have much, if any, impact on the performance of other code (a sleeping thread does just that—nothing at all). Of course, if all of those threads wake up at the same time and start hammering on the CPU, that's a different story—especially in light of the Global Interpreter Lock (GIL). Generally, you only want to use thread pools for I/O-bound processing.

One possible concern with creating large thread pools might be memory use. For example, if you create 2,000 threads on OS X, the system shows the Python process using up more than 9 GB of virtual memory. However, this is actually somewhat misleading. When creating a thread, the operating system reserves a region of virtual memory to hold the thread's execution stack (often as large as 8 MB). Only a small fragment of this memory is actually mapped to real memory, though. Thus, if you look a bit closer, you might find the Python process is using far less real memory (e.g., for 2,000 threads, only

70 MB of real memory is used, not 9 GB). If the size of the virtual memory is a concern, you can dial it down using the `threading.stack_size()` function. For example:

```
import threading
threading.stack_size(65536)
```

If you add this call and repeat the experiment of creating 2,000 threads, you'll find that the Python process is now only using about 210 MB of virtual memory, although the amount of real memory in use remains about the same. Note that the thread stack size must be at least 32,768 bytes, and is usually restricted to be a multiple of the system memory page size (4096, 8192, etc.).

## 12.8. Performing Simple Parallel Programming

### Problem

You have a program that performs a lot of CPU-intensive work, and you want to make it run faster by having it take advantage of multiple CPUs.

### Solution

The `concurrent.futures` library provides a `ProcessPoolExecutor` class that can be used to execute computationally intensive functions in a separately running instance of the Python interpreter. However, in order to use it, you first need to have some computationally intensive work. Let's illustrate with a simple yet practical example.

Suppose you have an entire directory of gzip-compressed Apache web server logs:

```
logs/
  20120701.log.gz
  20120702.log.gz
  20120703.log.gz
  20120704.log.gz
  20120705.log.gz
  20120706.log.gz
  ...
```

Further suppose each log file contains lines like this:

```
124.115.6.12 - - [10/Jul/2012:00:18:50 -0500] "GET /robots.txt ..." 200 71
210.212.209.67 - - [10/Jul/2012:00:18:51 -0500] "GET /ply/ ..." 200 11875
210.212.209.67 - - [10/Jul/2012:00:18:51 -0500] "GET /favicon.ico ..." 404 369
61.135.216.105 - - [10/Jul/2012:00:20:04 -0500] "GET /blog/atom.xml ..." 304 -
...
```

Here is a simple script that takes this data and identifies all hosts that have accessed the *robots.txt* file:

```

# findrobots.py

import gzip
import io
import glob

def find_robots(filename):
    """
    Find all of the hosts that access robots.txt in a single log file
    """
    robots = set()
    with gzip.open(filename) as f:
        for line in io.TextIOWrapper(f, encoding='ascii'):
            fields = line.split()
            if fields[6] == '/robots.txt':
                robots.add(fields[0])
    return robots

def find_all_robots(logdir):
    """
    Find all hosts across and entire sequence of files
    """
    files = glob.glob(logdir+'/*.log.gz')
    all_robots = set()
    for robots in map(find_robots, files):
        all_robots.update(robots)
    return all_robots

if __name__ == '__main__':
    robots = find_all_robots('logs')
    for ipaddr in robots:
        print(ipaddr)

```

The preceding program is written in the commonly used map-reduce style. The function `find_robots()` is mapped across a collection of filenames and the results are combined into a single result (the `all_robots` set in the `find_all_robots()` function).

Now, suppose you want to modify this program to use multiple CPUs. It turns out to be easy—simply replace the `map()` operation with a similar operation carried out on a process pool from the `concurrent.futures` library. Here is a slightly modified version of the code:

```

# findrobots.py

import gzip
import io
import glob
from concurrent import futures

def find_robots(filename):
    """
    Find all of the hosts that access robots.txt in a single log file

```



```

'''
robots = set()
with gzip.open(filename) as f:
    for line in io.TextIOWrapper(f,encoding='ascii'):
        fields = line.split()
        if fields[6] == '/robots.txt':
            robots.add(fields[0])
return robots

def find_all_robots(logdir):
'''
    Find all hosts across and entire sequence of files
'''
files = glob.glob(logdir+'/*.log.gz')
all_robots = set()
with futures.ProcessPoolExecutor() as pool:
    for robots in pool.map(find_robots, files):
        all_robots.update(robots)
return all_robots

if __name__ == '__main__':
    robots = find_all_robots('logs')
    for ipaddr in robots:
        print(ipaddr)

```

With this modification, the script produces the same result but runs about 3.5 times faster on our quad-core machine. The actual performance will vary according to the number of CPUs available on your machine.

## Discussion

Typical usage of a `ProcessPoolExecutor` is as follows:

```

from concurrent.futures import ProcessPoolExecutor

with ProcessPoolExecutor() as pool:
    ...
    do work in parallel using pool
    ...

```

Under the covers, a `ProcessPoolExecutor` creates *N* independent running Python interpreters where *N* is the number of available CPUs detected on the system. You can change the number of processes created by supplying an optional argument to `ProcessPoolExecutor(N)`. The pool runs until the last statement in the `with` block is executed, at which point the process pool is shut down. However, the program will wait until all submitted work has been processed.

Work to be submitted to a pool must be defined in a function. There are two methods for submission. If you are trying to parallelize a list comprehension or a `map()` operation, you use `pool.map()`:

```

# A function that performs a lot of work
def work(x):
    ...
    return result

# Nonparallel code
results = map(work, data)

# Parallel implementation
with ProcessPoolExecutor() as pool:
    results = pool.map(work, data)

```

Alternatively, you can manually submit single tasks using the `pool.submit()` method:

```

# Some function
def work(x):
    ...
    return result

with ProcessPoolExecutor() as pool:
    ...
    # Example of submitting work to the pool
    future_result = pool.submit(work, arg)

    # Obtaining the result (blocks until done)
    r = future_result.result()
    ...

```

If you manually submit a job, the result is an instance of `Future`. To obtain the actual result, you call its `result()` method. This blocks until the result is computed and returned by the pool.

Instead of blocking, you can also arrange to have a callback function triggered upon completion instead. For example:

```

def when_done(r):
    print('Got:', r.result())

with ProcessPoolExecutor() as pool:
    future_result = pool.submit(work, arg)
    future_result.add_done_callback(when_done)

```

The user-supplied callback function receives an instance of `Future` that must be used to obtain the actual result (i.e., by calling its `result()` method).

Although process pools can be easy to use, there are a number of important considerations to be made in designing larger programs. In no particular order:

- This technique for parallelization only works well for problems that can be trivially decomposed into independent parts.

- Work must be submitted in the form of simple functions. Parallel execution of instance methods, closures, or other kinds of constructs are not supported.
- Function arguments and return values must be compatible with `pickle`. Work is carried out in a separate interpreter using interprocess communication. Thus, data exchanged between interpreters has to be serialized.
- Functions submitted for work should not maintain persistent state or have side effects. With the exception of simple things such as logging, you don't really have any control over the behavior of child processes once started. Thus, to preserve your sanity, it is probably best to keep things simple and carry out work in pure-functions that don't alter their environment.
- Process pools are created by calling the `fork()` system call on Unix. This makes a clone of the Python interpreter, including all of the program state at the time of the fork. On Windows, an independent copy of the interpreter that does not clone state is launched. The actual forking process does not occur until the first `pool.map()` or `pool.submit()` method is called.
- Great care should be made when combining process pools and programs that use threads. In particular, you should probably create and launch process pools prior to the creation of any threads (e.g., create the pool in the main thread at program startup).

## 12.9. Dealing with the GIL (and How to Stop Worrying About It)

### Problem

You've heard about the Global Interpreter Lock (GIL), and are worried that it might be affecting the performance of your multithreaded program.

### Solution

Although Python fully supports thread programming, parts of the C implementation of the interpreter are not entirely thread safe to a level of allowing fully concurrent execution. In fact, the interpreter is protected by a so-called Global Interpreter Lock (GIL) that only allows one Python thread to execute at any given time. The most noticeable effect of the GIL is that multithreaded Python programs are not able to fully take advantage of multiple CPU cores (e.g., a computationally intensive application using more than one thread only runs on a single CPU).

Before discussing common GIL workarounds, it is important to emphasize that the GIL tends to only affect programs that are heavily CPU bound (i.e., dominated by computation). If your program is mostly doing I/O, such as network communication, threads are often a sensible choice because they're mostly going to spend their time sitting around waiting. In fact, you can create thousands of Python threads with barely a concern. Modern operating systems have no trouble running with that many threads, so it's simply not something you should worry much about.

For CPU-bound programs, you really need to study the nature of the computation being performed. For instance, careful choice of the underlying algorithm may produce a far greater speedup than trying to parallelize an unoptimal algorithm with threads. Similarly, given that Python is interpreted, you might get a far greater speedup simply by moving performance-critical code into a C extension module. Extensions such as **NumPy** are also highly effective at speeding up certain kinds of calculations involving array data. Last, but not least, you might investigate alternative implementations, such as PyPy, which features optimizations such as a JIT compiler (although, as of this writing, it does not yet support Python 3).

It's also worth noting that threads are not necessarily used exclusively for performance. A CPU-bound program might be using threads to manage a graphical user interface, a network connection, or provide some other kind of service. In this case, the GIL can actually present more of a problem, since code that holds it for an excessively long period will cause annoying stalls in the non-CPU-bound threads. In fact, a poorly written C extension can actually make this problem worse, even though the computation part of the code might run faster than before.

Having said all of this, there are two common strategies for working around the limitations of the GIL. First, if you are working entirely in Python, you can use the `multiprocessing` module to create a process pool and use it like a co-processor. For example, suppose you have the following thread code:

```
# Performs a large calculation (CPU bound)
def some_work(args):
    ...
    return result

# A thread that calls the above function
def some_thread():
    while True:
        ...
        r = some_work(args)
        ...
```

Here's how you would modify the code to use a pool:

```
# Processing pool (see below for initialization)
pool = None
```

```

# Performs a large calculation (CPU bound)
def some_work(args):
    ...
    return result

# A thread that calls the above function
def some_thread():
    while True:
        ...
        r = pool.apply(some_work, (args))
        ...

# Initiaze the pool
if __name__ == '__main__':
    import multiprocessing
    pool = multiprocessing.Pool()

```

This example with a pool works around the GIL using a neat trick. Whenever a thread wants to perform CPU-intensive work, it hands the work to the pool. The pool, in turn, hands the work to a separate Python interpreter running in a different process. While the thread is waiting for the result, it releases the GIL. Moreover, because the calculation is being performed in a separate interpreter, it's no longer bound by the restrictions of the GIL. On a multicore system, you'll find that this technique easily allows you to take advantage of all the CPUs.

The second strategy for working around the GIL is to focus on C extension programming. The general idea is to move computationally intensive tasks to C, independent of Python, and have the C code release the GIL while it's working. This is done by inserting special macros into the C code like this:

```

#include "Python.h"
...

PyObject *pyfunc(PyObject *self, PyObject *args) {
    ...
    Py_BEGIN_ALLOW_THREADS
    // Threaded C code
    ...
    Py_END_ALLOW_THREADS
    ...
}

```

If you are using other tools to access C, such as the ctypes library or Cython, you may not need to do anything. For example, ctypes releases the GIL when calling into C by default.

## Discussion

Many programmers, when faced with thread performance problems, are quick to blame the GIL for all of their ills. However, doing so is shortsighted and naive. Just as a real-

world example, mysterious “stalls” in a multithreaded network program might be caused by something entirely different (e.g., a stalled DNS lookup) rather than anything related to the GIL. The bottom line is that you really need to study your code to know if the GIL is an issue or not. Again, realize that the GIL is mostly concerned with CPU-bound processing, not I/O.

If you are going to use a process pool as a workaround, be aware that doing so involves data serialization and communication with a different Python interpreter. For this to work, the operation to be performed needs to be contained within a Python function defined by the `def` statement (i.e., no lambdas, closures, callable instances, etc.), and the function arguments and return value must be compatible with `pickle`. Also, the amount of work to be performed must be sufficiently large to make up for the extra communication overhead.

Another subtle aspect of pools is that mixing threads and process pools together can be a good way to make your head explode. If you are going to use both of these features together, it is often best to create the process pool as a singleton at program startup, prior to the creation of any threads. Threads will then use the same process pool for all of their computationally intensive work.

For C extensions, the most important feature is maintaining isolation from the Python interpreter process. That is, if you’re going to offload work from Python to C, you need to make sure the C code operates independently of Python. This means using no Python data structures and making no calls to Python’s C API. Another consideration is that you want to make sure your C extension does enough work to make it all worthwhile. That is, it’s much better if the extension can perform millions of calculations as opposed to just a few small calculations.

Needless to say, these solutions to working around the GIL don’t apply to all possible problems. For instance, certain kinds of applications don’t work well if separated into multiple processes, nor may you want to code parts in C. For these kinds of applications, you may have to come up with your own solution (e.g., multiple processes accessing shared memory regions, multiple interpreters running in the same process, etc.). Alternatively, you might look at some other implementations of the interpreter, such as PyPy.

See Recipes [15.7](#) and [15.10](#) for additional information on releasing the GIL in C extensions.

## 12.10. Defining an Actor Task

### Problem

You’d like to define tasks with behavior similar to “actors” in the so-called “actor model.”

## Solution

The “actor model” is one of the oldest and most simple approaches to concurrency and distributed computing. In fact, its underlying simplicity is part of its appeal. In a nutshell, an actor is a concurrently executing task that simply acts upon messages sent to it. In response to these messages, it may decide to send further messages to other actors. Communication with actors is one way and asynchronous. Thus, the sender of a message does not know when a message actually gets delivered, nor does it receive a response or acknowledgment that the message has been processed.

Actors are straightforward to define using a combination of a thread and a queue. For example:

```
from queue import Queue
from threading import Thread, Event

# Sentinel used for shutdown
class ActorExit(Exception):
    pass

class Actor:
    def __init__(self):
        self._mailbox = Queue()

    def send(self, msg):
        """
        Send a message to the actor
        """
        self._mailbox.put(msg)

    def recv(self):
        """
        Receive an incoming message
        """
        msg = self._mailbox.get()
        if msg is ActorExit:
            raise ActorExit()
        return msg

    def close(self):
        """
        Close the actor, thus shutting it down
        """
        self.send(ActorExit)

    def start(self):
        """
        Start concurrent execution
        """
        self._terminated = Event()
        t = Thread(target=self._bootstrap)
```

```

        t.daemon = True
        t.start()

    def _bootstrap(self):
        try:
            self.run()
        except ActorExit:
            pass
        finally:
            self._terminated.set()

    def join(self):
        self._terminated.wait()

    def run(self):
        """
        Run method to be implemented by the user
        """
        while True:
            msg = self.recv()

# Sample ActorTask
class PrintActor(Actor):
    def run(self):
        while True:
            msg = self.recv()
            print('Got:', msg)

# Sample use
p = PrintActor()
p.start()
p.send('Hello')
p.send('World')
p.close()
p.join()

```

In this example, Actor instances are things that you simply send a message to using their `send()` method. Under the covers, this places the message on a queue and hands it off to an internal thread that runs to process the received messages. The `close()` method is programmed to shut down the actor by placing a special sentinel value (`ActorExit`) on the queue. Users define new actors by inheriting from `Actor` and re-defining the `run()` method to implement their custom processing. The usage of the `ActorExit` exception is such that user-defined code can be programmed to catch the termination request and handle it if appropriate (the exception is raised by the `get()` method and propagated).

If you relax the requirement of concurrent and asynchronous message delivery, actor-like objects can also be minimally defined by generators. For example:

```

def print_actor():
    while True:

```



```

try:
    msg = yield          # Get a message
    print('Got:', msg)
except GeneratorExit:
    print('Actor terminating')

# Sample use
p = print_actor()
next(p)                # Advance to the yield (ready to receive)
p.send('Hello')
p.send('World')
p.close()

```

## Discussion

Part of the appeal of actors is their underlying simplicity. In practice, there is just one core operation, `send()`. Plus, the general concept of a “message” in actor-based systems is something that can be expanded in many different directions. For example, you could pass tagged messages in the form of tuples and have actors take different courses of action like this:

```

class TaggedActor(Actor):
    def run(self):
        while True:
            tag, *payload = self.recv()
            getattr(self, 'do_' + tag)(*payload)

# Methods corresponding to different message tags
def do_A(self, x):
    print('Running A', x)

def do_B(self, x, y):
    print('Running B', x, y)

# Example
a = TaggedActor()
a.start()
a.send(('A', 1))    # Invokes do_A(1)
a.send(('B', 2, 3)) # Invokes do_B(2,3)

```

As another example, here is a variation of an actor that allows arbitrary functions to be executed in a worker and results to be communicated back using a special `Result` object:

```

from threading import Event
class Result:
    def __init__(self):
        self._evt = Event()
        self._result = None

    def set_result(self, value):
        self._result = value

```

```

        self._evt.set()

    def result(self):
        self._evt.wait()
        return self._result

class Worker(Actor):
    def submit(self, func, *args, **kwargs):
        r = Result()
        self.send((func, args, kwargs, r))
        return r

    def run(self):
        while True:
            func, args, kwargs, r = self.recv()
            r.set_result(func(*args, **kwargs))

# Example use
worker = Worker()
worker.start()
r = worker.submit(pow, 2, 3)
print(r.result())

```

Last, but not least, the concept of “sending” a task a message is something that can be scaled up into systems involving multiple processes or even large distributed systems. For example, the `send()` method of an actor-like object could be programmed to transmit data on a socket connection or deliver it via some kind of messaging infrastructure (e.g., AMQP, ZMQ, etc.).

## 12.11. Implementing Publish/Subscribe Messaging

### Problem

You have a program based on communicating threads and want them to implement publish/subscribe messaging.

### Solution

To implement publish/subscribe messaging, you typically introduce a separate “exchange” or “gateway” object that acts as an intermediary for all messages. That is, instead of directly sending a message from one task to another, a message is sent to the exchange and it delivers it to one or more attached tasks. Here is one example of a very simple exchange implementation:

```

from collections import defaultdict

class Exchange:
    def __init__(self):
        self._subscribers = set()

```

```

def attach(self, task):
    self._subscribers.add(task)

def detach(self, task):
    self._subscribers.remove(task)

def send(self, msg):
    for subscriber in self._subscribers:
        subscriber.send(msg)

# Dictionary of all created exchanges
_exchanges = defaultdict(Exchange)

# Return the Exchange instance associated with a given name
def get_exchange(name):
    return _exchanges[name]

```

An exchange is really nothing more than an object that keeps a set of active subscribers and provides methods for attaching, detaching, and sending messages. Each exchange is identified by a name, and the `get_exchange()` function simply returns the Exchange instance associated with a given name.

Here is a simple example that shows how to use an exchange:

```

# Example of a task. Any object with a send() method

class Task:
    ...
    def send(self, msg):
        ...

task_a = Task()
task_b = Task()

# Example of getting an exchange
exc = get_exchange('name')

# Examples of subscribing tasks to it
exc.attach(task_a)
exc.attach(task_b)

# Example of sending messages
exc.send('msg1')
exc.send('msg2')

# Example of unsubscribing
exc.detach(task_a)
exc.detach(task_b)

```

Although there are many different variations on this theme, the overall idea is the same. Messages will be delivered to an exchange and the exchange will deliver them to attached subscribers.

## Discussion

The concept of tasks or threads sending messages to one another (often via queues) is easy to implement and quite popular. However, the benefits of using a public/subscribe (pub/sub) model instead are often overlooked.

First, the use of an exchange can simplify much of the plumbing involved in setting up communicating threads. Instead of trying to wire threads together across multiple program modules, you only worry about connecting them to a known exchange. In some sense, this is similar to how the `logging` library works. In practice, it can make it easier to decouple various tasks in the program.

Second, the ability of the exchange to broadcast messages to multiple subscribers opens up new communication patterns. For example, you could implement systems with redundant tasks, broadcasting, or fan-out. You could also build debugging and diagnostic tools that attach themselves to exchanges as ordinary subscribers. For example, here is a simple diagnostic class that would display sent messages:

```
class DisplayMessages:
    def __init__(self):
        self.count = 0
    def send(self, msg):
        self.count += 1
        print('msg[{}]: {}'.format(self.count, msg))

exc = get_exchange('name')
d = DisplayMessages()
exc.attach(d)
```

Last, but not least, a notable aspect of the implementation is that it works with a variety of task-like objects. For example, the receivers of a message could be actors (as described in [Recipe 12.10](#)), coroutines, network connections, or just about anything that implements a proper `send()` method.

One potentially problematic aspect of an exchange concerns the proper attachment and detachment of subscribers. In order to properly manage resources, every subscriber that attaches must eventually detach. This leads to a programming model similar to this:

```
exc = get_exchange('name')
exc.attach(some_task)
try:
    ...
finally:
    exc.detach(some_task)
```

In some sense, this is similar to the usage of files, locks, and similar objects. Experience has shown that it is quite easy to forget the final `detach()` step. To simplify this, you might consider the use of the context-management protocol. For example, adding a `subscribe()` method to the exchange like this:

```
from contextlib import contextmanager
from collections import defaultdict

class Exchange:
    def __init__(self):
        self._subscribers = set()

    def attach(self, task):
        self._subscribers.add(task)

    def detach(self, task):
        self._subscribers.remove(task)

    @contextmanager
    def subscribe(self, *tasks):
        for task in tasks:
            self.attach(task)
        try:
            yield
        finally:
            for task in tasks:
                self.detach(task)

    def send(self, msg):
        for subscriber in self._subscribers:
            subscriber.send(msg)

# Dictionary of all created exchanges
_exchanges = defaultdict(Exchange)

# Return the Exchange instance associated with a given name
def get_exchange(name):
    return _exchanges[name]

# Example of using the subscribe() method
exc = get_exchange('name')
with exc.subscribe(task_a, task_b):
    ...
    exc.send('msg1')
    exc.send('msg2')
    ...

# task_a and task_b detached here
```

Finally, it should be noted that there are numerous possible extensions to the exchange idea. For example, exchanges could implement an entire collection of message channels

or apply pattern matching rules to exchange names. Exchanges can also be extended into distributed computing applications (e.g., routing messages to tasks on different machines, etc.).

## 12.12. Using Generators As an Alternative to Threads

### Problem

You want to implement concurrency using generators (coroutines) as an alternative to system threads. This is sometimes known as user-level threading or green threading.

### Solution

To implement your own concurrency using generators, you first need a fundamental insight concerning generator functions and the `yield` statement. Specifically, the fundamental behavior of `yield` is that it causes a generator to suspend its execution. By suspending execution, it is possible to write a scheduler that treats generators as a kind of “task” and alternates their execution using a kind of cooperative task switching.

To illustrate this idea, consider the following two generator functions using a simple `yield`:

```
# Two simple generator functions
def countdown(n):
    while n > 0:
        print('T-minus', n)
        yield
        n -= 1
    print('Blastoff!')

def countup(n):
    x = 0
    while x < n:
        print('Counting up', x)
        yield
        x += 1
```

These functions probably look a bit funny using `yield` all by itself. However, consider the following code that implements a simple task scheduler:

```
from collections import deque

class TaskScheduler:
    def __init__(self):
        self._task_queue = deque()

    def new_task(self, task):
        '''
        Admit a newly started task to the scheduler
        '''
```

```

    '''
    self._task_queue.append(task)

def run(self):
    '''
    Run until there are no more tasks
    '''
    while self._task_queue:
        task = self._task_queue.popleft()
        try:
            # Run until the next yield statement
            next(task)
            self._task_queue.append(task)
        except StopIteration:
            # Generator is no longer executing
            pass

# Example use
sched = TaskScheduler()
sched.new_task(countdown(10))
sched.new_task(countdown(5))
sched.new_task(countup(15))
sched.run()

```

In this code, the `TaskScheduler` class runs a collection of generators in a round-robin manner—each one running until they reach a `yield` statement. For the sample, the output will be as follows:

```

T-minus 10
T-minus 5
Counting up 0
T-minus 9
T-minus 4
Counting up 1
T-minus 8
T-minus 3
Counting up 2
T-minus 7
T-minus 2
...

```

At this point, you’ve essentially implemented the tiny core of an “operating system” if you will. Generator functions are the tasks and the `yield` statement is how tasks signal that they want to suspend. The scheduler simply cycles over the tasks until none are left executing.

In practice, you probably wouldn’t use generators to implement concurrency for something as simple as shown. Instead, you might use generators to replace the use of threads when implementing actors (see [Recipe 12.10](#)) or network servers.

The following code illustrates the use of generators to implement a thread-free version of actors:

```
from collections import deque

class ActorScheduler:
    def __init__(self):
        self._actors = { }           # Mapping of names to actors
        self._msg_queue = deque()    # Message queue

    def new_actor(self, name, actor):
        """
        Admit a newly started actor to the scheduler and give it a name
        """
        self._msg_queue.append((actor, None))
        self._actors[name] = actor

    def send(self, name, msg):
        """
        Send a message to a named actor
        """
        actor = self._actors.get(name)
        if actor:
            self._msg_queue.append((actor, msg))

    def run(self):
        """
        Run as long as there are pending messages.
        """
        while self._msg_queue:
            actor, msg = self._msg_queue.popleft()
            try:
                actor.send(msg)
            except StopIteration:
                pass

# Example use
if __name__ == '__main__':
    def printer():
        while True:
            msg = yield
            print('Got:', msg)

    def counter(sched):
        while True:
            # Receive the current count
            n = yield
            if n == 0:
                break
            # Send to the printer task
            sched.send('printer', n)
            # Send the next count to the counter task (recursive)
```



```

        sched.send('counter', n-1)

    sched = ActorScheduler()
    # Create the initial actors
    sched.new_actor('printer', printer())
    sched.new_actor('counter', counter(sched))

    # Send an initial message to the counter to initiate
    sched.send('counter', 10000)
    sched.run()

```

The execution of this code might take a bit of study, but the key is the queue of pending messages. Essentially, the scheduler runs as long as there are messages to deliver. A remarkable feature is that the counter generator sends messages to itself and ends up in a recursive cycle not bound by Python's recursion limit.

Here is an advanced example showing the use of generators to implement a concurrent network application:

```

from collections import deque
from select import select

# This class represents a generic yield event in the scheduler
class YieldEvent:
    def handle_yield(self, sched, task):
        pass
    def handle_resume(self, sched, task):
        pass

# Task Scheduler
class Scheduler:
    def __init__(self):
        self._numtasks = 0      # Total num of tasks
        self._ready = deque()   # Tasks ready to run
        self._read_waiting = {} # Tasks waiting to read
        self._write_waiting = {} # Tasks waiting to write

    # Poll for I/O events and restart waiting tasks
    def _iopoll(self):
        rset, wset, eset = select(self._read_waiting,
                                   self._write_waiting, [])

        for r in rset:
            evt, task = self._read_waiting.pop(r)
            evt.handle_resume(self, task)
        for w in wset:
            evt, task = self._write_waiting.pop(w)
            evt.handle_resume(self, task)

    def new(self, task):
        '''
        Add a newly started task to the scheduler
        '''

```

```

        self._ready.append((task, None))
        self._numtasks += 1

    def add_ready(self, task, msg=None):
        """
        Append an already started task to the ready queue.
        msg is what to send into the task when it resumes.
        """
        self._ready.append((task, msg))

    # Add a task to the reading set
    def _read_wait(self, fileno, evt, task):
        self._read_waiting[fileno] = (evt, task)

    # Add a task to the write set
    def _write_wait(self, fileno, evt, task):
        self._write_waiting[fileno] = (evt, task)

    def run(self):
        """
        Run the task scheduler until there are no tasks
        """
        while self._numtasks:
            if not self._ready:
                self._iopoll()
            task, msg = self._ready.popleft()
            try:
                # Run the coroutine to the next yield
                r = task.send(msg)
                if isinstance(r, YieldEvent):
                    r.handle_yield(self, task)
                else:
                    raise RuntimeError('unrecognized yield event')
            except StopIteration:
                self._numtasks -= 1

    # Example implementation of coroutine-based socket I/O
    class ReadSocket(YieldEvent):
        def __init__(self, sock, nbytes):
            self.sock = sock
            self.nbytes = nbytes
        def handle_yield(self, sched, task):
            sched._read_wait(self.sock.fileno(), self, task)
        def handle_resume(self, sched, task):
            data = self.sock.recv(self.nbytes)
            sched.add_ready(task, data)

    class WriteSocket(YieldEvent):
        def __init__(self, sock, data):
            self.sock = sock
            self.data = data
        def handle_yield(self, sched, task):

```

```

        sched._write_wait(self.sock.fileno(), self, task)
    def handle_resume(self, sched, task):
        nsent = self.sock.send(self.data)
        sched.add_ready(task, nsent)

class AcceptSocket(YieldEvent):
    def __init__(self, sock):
        self.sock = sock
    def handle_yield(self, sched, task):
        sched._read_wait(self.sock.fileno(), self, task)
    def handle_resume(self, sched, task):
        r = self.sock.accept()
        sched.add_ready(task, r)

# Wrapper around a socket object for use with yield
class Socket(object):
    def __init__(self, sock):
        self._sock = sock
    def recv(self, maxbytes):
        return ReadSocket(self._sock, maxbytes)
    def send(self, data):
        return WriteSocket(self._sock, data)
    def accept(self):
        return AcceptSocket(self._sock)
    def __getattr__(self, name):
        return getattr(self._sock, name)

if __name__ == '__main__':
    from socket import socket, AF_INET, SOCK_STREAM
    import time

    # Example of a function involving generators. This should
    # be called using line = yield from readline(sock)
    def readline(sock):
        chars = []
        while True:
            c = yield sock.recv(1)
            if not c:
                break
            chars.append(c)
            if c == b'\n':
                break
        return b''.join(chars)

    # Echo server using generators
    class EchoServer:
        def __init__(self, addr, sched):
            self.sched = sched
            sched.new(self.server_loop(addr))

        def server_loop(self, addr):
            s = Socket(socket(AF_INET, SOCK_STREAM))

```

```

s.bind(addr)
s.listen(5)
while True:
    c,a = yield s.accept()
    print('Got connection from ', a)
    self.sched.new(self.client_handler(Socket(c)))

def client_handler(self,client):
    while True:
        line = yield from readline(client)
        if not line:
            break
        line = b'GOT:' + line
        while line:
            nsent = yield client.send(line)
            line = line[nsent:]
        client.close()
        print('Client closed')

sched = Scheduler()
EchoServer('',16000),sched)
sched.run()

```

This code will undoubtedly require a certain amount of careful study. However, it is essentially implementing a small operating system. There is a queue of tasks ready to run and there are waiting areas for tasks sleeping for I/O. Much of the scheduler involves moving tasks between the ready queue and the I/O waiting area.

## Discussion

When building generator-based concurrency frameworks, it is most common to work with the more general form of `yield`:

```

def some_generator():
    ...
    result = yield data
    ...

```

Functions that use `yield` in this manner are more generally referred to as “coroutines.” Within a scheduler, the `yield` statement gets handled in a loop as follows:

```

f = some_generator()

# Initial result. Is None to start since nothing has been computed
result = None
while True:
    try:
        data = f.send(result)
        result = ... do some calculation ...
    except StopIteration:
        break

```

The logic concerning the `result` is a bit convoluted. However, the value passed to `send()` defines what gets returned when the `yield` statement wakes back up. So, if a `yield` is going to return a result in response to data that was previously yielded, it gets returned on the next `send()` operation. If a generator function has just started, sending in a value of `None` simply makes it advance to the first `yield` statement.

In addition to sending in values, it is also possible to execute a `close()` method on a generator. This causes a silent `GeneratorExit` exception to be raised at the `yield` statement, which stops execution. If desired, a generator can catch this exception and perform cleanup actions. It's also possible to use the `throw()` method of a generator to raise an arbitrary exception at the `yield` statement. A task scheduler might use this to communicate errors into running generators.

The `yield from` statement used in the last example is used to implement coroutines that serve as subroutines or procedures to be called from other generators. Essentially, control transparently transfers to the new function. Unlike normal generators, a function that is called using `yield from` can return a value that becomes the result of the `yield from` statement. More information about `yield from` can be found in [PEP 380](#).

Finally, if programming with generators, it is important to stress that there are some major limitations. In particular, you get none of the benefits that threads provide. For instance, if you execute any code that is CPU bound or which blocks for I/O, it will suspend the entire task scheduler until the completion of that operation. To work around this, your only real option is to delegate the operation to a separate thread or process where it can run independently. Another limitation is that most Python libraries have not been written to work well with generator-based threading. If you take this approach, you may find that you need to write replacements for many standard library functions.

As basic background on coroutines and the techniques utilized in this recipe, see [PEP 342](#) and “[A Curious Course on Coroutines and Concurrency](#)”.

[PEP 3156](#) also has a modern take on asynchronous I/O involving coroutines. In practice, it is extremely unlikely that you will write a low-level coroutine scheduler yourself. However, ideas surrounding coroutines are the basis for many popular libraries, including [gevent](#), [greenlet](#), [Stackless Python](#), and similar projects.

## 12.13. Polling Multiple Thread Queues

### Problem

You have a collection of thread queues, and you would like to be able to poll them for incoming items, much in the same way as you might poll a collection of network connections for incoming data.

## Solution

A common solution to polling problems involves a little-known trick involving a hidden loopback network connection. Essentially, the idea is as follows: for each queue (or any object) that you want to poll, you create a pair of connected sockets. You then write on one of the sockets to signal the presence of data. The other socket is then passed to `select()` or a similar function to poll for the arrival of data. Here is some sample code that illustrates this idea:

```
import queue
import socket
import os

class PollableQueue(queue.Queue):
    def __init__(self):
        super().__init__()
        # Create a pair of connected sockets
        if os.name == 'posix':
            self._putsocket, self._getsocket = socket.socketpair()
        else:
            # Compatibility on non-POSIX systems
            server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            server.bind(('127.0.0.1', 0))
            server.listen(1)
            self._putsocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            self._putsocket.connect(server.getsockname())
            self._getsocket, _ = server.accept()
            server.close()

    def fileno(self):
        return self._getsocket.fileno()

    def put(self, item):
        super().put(item)
        self._putsocket.send(b'x')

    def get(self):
        self._getsocket.recv(1)
        return super().get()
```

In this code, a new kind of Queue instance is defined where there is an underlying pair of connected sockets. The `socketpair()` function on Unix machines can establish such sockets easily. On Windows, you have to fake it using code similar to that shown (it looks a bit weird, but a server socket is created and a client immediately connects to it afterward). The normal `get()` and `put()` methods are then redefined slightly to perform a small bit of I/O on these sockets. The `put()` method writes a single byte of data to one of the sockets after putting data on the queue. The `get()` method reads a single byte of data from the other socket when removing an item from the queue.

The `fileno()` method is what makes the queue pollable using a function such as `select()`. Essentially, it just exposes the underlying file descriptor of the socket used by the `get()` function.

Here is an example of some code that defines a consumer which monitors multiple queues for incoming items:

```
import select
import threading

def consumer(queues):
    """
    Consumer that reads data on multiple queues simultaneously
    """
    while True:
        can_read, _, _ = select.select(queues,[],[])
        for r in can_read:
            item = r.get()
            print('Got:', item)

q1 = PollableQueue()
q2 = PollableQueue()
q3 = PollableQueue()
t = threading.Thread(target=consumer, args=(q1,q2,q3,))
t.daemon = True
t.start()

# Feed data to the queues
q1.put(1)
q2.put(10)
q3.put('hello')
q2.put(15)
...
```

If you try it, you'll find that the consumer indeed receives all of the put items, regardless of which queues they are placed in.

## Discussion

The problem of polling non-file-like objects, such as queues, is often a lot trickier than it looks. For instance, if you don't use the socket technique shown, your only option is to write code that cycles through the queues and uses a timer, like this:

```
import time

def consumer(queues):
    while True:
        for q in queues:
            if not q.empty():
                item = q.get()
                print('Got:', item)
```

```
# Sleep briefly to avoid 100% CPU
time.sleep(0.01)
```

This might work for certain kinds of problems, but it's clumsy and introduces other weird performance problems. For example, if new data is added to a queue, it won't be detected for as long as 10 milliseconds (an eternity on a modern processor).

You run into even further problems if the preceding polling is mixed with the polling of other objects, such as network sockets. For example, if you want to poll both sockets and queues at the same time, you might have to use code like this:

```
import select

def event_loop(sockets, queues):
    while True:
        # polling with a timeout
        can_read, _, _ = select.select(sockets, [], [], 0.01)
        for r in can_read:
            handle_read(r)
        for q in queues:
            if not q.empty():
                item = q.get()
                print('Got:', item)
```

The solution shown solves a lot of these problems by simply putting queues on equal status with sockets. A single `select()` call can be used to poll for activity on both. It is not necessary to use timeouts or other time-based hacks to periodically check. Moreover, if data gets added to a queue, the consumer will be notified almost instantaneously. Although there is a tiny amount of overhead associated with the underlying I/O, it often is worth it to have better response time and simplified coding.

## 12.14. Launching a Daemon Process on Unix

### Problem

You would like to write a program that runs as a proper daemon process on Unix or Unix-like systems.

### Solution

Creating a proper daemon process requires a precise sequence of system calls and careful attention to detail. The following code shows how to define a daemon process along with the ability to easily stop it once launched:

```
#!/usr/bin/env python3
# daemon.py

import os
import sys
```



```

import atexit
import signal

def daemonize(pidfile, *, stdin='/dev/null',
               stdout='/dev/null',
               stderr='/dev/null'):

    if os.path.exists(pidfile):
        raise RuntimeError('Already running')

    # First fork (detaches from parent)
    try:
        if os.fork() > 0:
            raise SystemExit(0) # Parent exit
    except OSError as e:
        raise RuntimeError('fork #1 failed.')

    os.chdir('/')
    os.umask(0)
    os.setsid()
    # Second fork (relinquish session leadership)
    try:
        if os.fork() > 0:
            raise SystemExit(0)
    except OSError as e:
        raise RuntimeError('fork #2 failed.')

    # Flush I/O buffers
    sys.stdout.flush()
    sys.stderr.flush()

    # Replace file descriptors for stdin, stdout, and stderr
    with open(stdin, 'rb', 0) as f:
        os.dup2(f.fileno(), sys.stdin.fileno())
    with open(stdout, 'ab', 0) as f:
        os.dup2(f.fileno(), sys.stdout.fileno())
    with open(stderr, 'ab', 0) as f:
        os.dup2(f.fileno(), sys.stderr.fileno())

    # Write the PID file
    with open(pidfile, 'w') as f:
        print(os.getpid(), file=f)

    # Arrange to have the PID file removed on exit/signal
    atexit.register(lambda: os.remove(pidfile))

    # Signal handler for termination (required)
    def sigterm_handler(signo, frame):
        raise SystemExit(1)

    signal.signal(signal.SIGTERM, sigterm_handler)

```

```

def main():
    import time
    sys.stdout.write('Daemon started with pid {}\n'.format(os.getpid()))
    while True:
        sys.stdout.write('Daemon Alive! {}\n'.format(time.ctime()))
        time.sleep(10)

if __name__ == '__main__':
    PIDFILE = '/tmp/daemon.pid'

    if len(sys.argv) != 2:
        print('Usage: {} [start|stop]'.format(sys.argv[0]), file=sys.stderr)
        raise SystemExit(1)

    if sys.argv[1] == 'start':
        try:
            daemonize(PIDFILE,
                      stdout='/tmp/daemon.log',
                      stderr='/tmp/dameon.log')
        except RuntimeError as e:
            print(e, file=sys.stderr)
            raise SystemExit(1)

    main()

    elif sys.argv[1] == 'stop':
        if os.path.exists(PIDFILE):
            with open(PIDFILE) as f:
                os.kill(int(f.read()), signal.SIGTERM)
        else:
            print('Not running', file=sys.stderr)
            raise SystemExit(1)

    else:
        print('Unknown command {}'.format(sys.argv[1]), file=sys.stderr)
        raise SystemExit(1)

```

To launch the daemon, the user would use a command like this:

```

bash % daemon.py start
bash % cat /tmp/daemon.pid
2882
bash % tail -f /tmp/daemon.log
Daemon started with pid 2882
Daemon Alive! Fri Oct 12 13:45:37 2012
Daemon Alive! Fri Oct 12 13:45:47 2012
...

```

Daemon processes run entirely in the background, so the command returns immediately. However, you can view its associated pid file and log, as just shown. To stop the daemon, use:

```
bash % daemon.py stop
bash %
```

## Discussion

This recipe defines a function `daemonize()` that should be called at program startup to make the program run as a daemon. The signature to `daemonize()` is using keyword-only arguments to make the purpose of the optional arguments more clear when used. This forces the user to use a call such as this:

```
daemonize('daemon.pid',
          stdin='/dev/null',
          stdout='/tmp/daemon.log',
          stderr='/tmp/daemon.log')
```

As opposed to a more cryptic call such as:

```
# Illegal. Must use keyword arguments
daemonize('daemon.pid',
          '/dev/null', '/tmp/daemon.log', '/tmp/daemon.log')
```

The steps involved in creating a daemon are fairly cryptic, but the general idea is as follows. First, a daemon has to detach itself from its parent process. This is the purpose of the first `os.fork()` operation and immediate termination by the parent.

After the child has been orphaned, the call to `os.setsid()` creates an entirely new process session and sets the child as the leader. This also sets the child as the leader of a new process group and makes sure there is no controlling terminal. If this all sounds a bit too magical, it has to do with getting the daemon to detach properly from the terminal and making sure that things like signals don't interfere with its operation.

The calls to `os.chdir()` and `os.umask(0)` change the current working directory and reset the file mode mask. Changing the directory is usually a good idea so that the daemon is no longer working in the directory from which it was launched.

The second call to `os.fork()` is by far the more mysterious operation here. This step makes the daemon process give up the ability to acquire a new controlling terminal and provides even more isolation (essentially, the daemon gives up its session leadership and thus no longer has the permission to open controlling terminals). Although you could probably omit this step, it's typically recommended.

Once the daemon process has been properly detached, it performs steps to reinitialize the standard I/O streams to point at files specified by the user. This part is actually somewhat tricky. References to file objects associated with the standard I/O streams are found in multiple places in the interpreter (`sys.stdout`, `sys.__stdout__`, etc.). Simply closing `sys.stdout` and reassigning it is not likely to work correctly, because there's no way to know if it will fix all uses of `sys.stdout`. Instead, a separate file object is opened, and the `os.dup2()` call is used to have it replace the file descriptor currently being used

by `sys.stdout`. When this happens, the original file for `sys.stdout` will be closed and the new one takes its place. It must be emphasized that any file encoding or text handling already applied to the standard I/O streams will remain in place.

A common practice with daemon processes is to write the process ID of the daemon in a file for later use by other programs. The last part of the `daemonize()` function writes this file, but also arranges to have the file removed on program termination. The `atexit.register()` function registers a function to execute when the Python interpreter terminates. The definition of a signal handler for `SIGTERM` is also required for a graceful termination. The signal handler merely raises `SystemExit()` and nothing more. This might look unnecessary, but without it, termination signals kill the interpreter without performing the cleanup actions registered with `atexit.register()`. An example of code that kills the daemon can be found in the handling of the stop command at the end of the program.

More information about writing daemon processes can be found in *Advanced Programming in the UNIX Environment*, 2nd Edition, by W. Richard Stevens and Stephen A. Rago (Addison-Wesley, 2005). Although focused on C programming, all of the material is easily adapted to Python, since all of the required POSIX functions are available in the standard library.

---

# Utility Scripting and System Administration

A lot of people use Python as a replacement for shell scripts, using it to automate common system tasks, such as manipulating files, configuring systems, and so forth. The main goal of this chapter is to describe features related to common tasks encountered when writing scripts. For example, parsing command-line options, manipulating files on the filesystem, getting useful system configuration data, and so forth. [Chapter 5](#) also contains general information related to files and directories.

## 13.1. Accepting Script Input via Redirection, Pipes, or Input Files

### Problem

You want a script you've written to be able to accept input using whatever mechanism is easiest for the user. This should include piping output from a command to the script, redirecting a file into the script, or just passing a filename, or list of filenames, to the script on the command line.

### Solution

Python's built-in `fileinput` module makes this very simple and concise. If you have a script that looks like this:

```
#!/usr/bin/env python3
import fileinput

with fileinput.input() as f_input:
    for line in f_input:
        print(line, end='')
```

Then you can already accept input to the script in all of the previously mentioned ways. If you save this script as *filein.py* and make it executable, you can do all of the following and get the expected output:

```
$ ls | ./filein.py           # Prints a directory listing to stdout.
$ ./filein.py /etc/passwd   # Reads /etc/passwd to stdout.
$ ./filein.py < /etc/passwd # Reads /etc/passwd to stdout.
```

## Discussion

The `fileinput.input()` function creates and returns an instance of the `FileInput` class. In addition to containing a few handy helper methods, the instance can also be used as a context manager. So, to put all of this together, if we wrote a script that expected to be printing output from several files at once, we might have it include the filename and line number in the output, like this:

```
>>> import fileinput
>>> with fileinput.input('/etc/passwd') as f:
>>>     for line in f:
...         print(f.filename(), f.lineno(), line, end='')
...
/etc/passwd 1 ##
/etc/passwd 2 # User Database
/etc/passwd 3 #

<other output omitted>
```

Using it as a context manager ensures that the file is closed when it's no longer being used, and we leveraged a few handy `FileInput` helper methods here to get some extra information in the output.

## 13.2. Terminating a Program with an Error Message

### Problem

You want your program to terminate by printing a message to standard error and returning a nonzero status code.

### Solution

To have a program terminate in this manner, raise a `SystemExit` exception, but supply the error message as an argument. For example:

```
raise SystemExit('It failed!')
```

This will cause the supplied message to be printed to `sys.stderr` and the program to exit with a status code of 1.

## Discussion

This is a small recipe, but it solves a common problem that arises when writing scripts. Namely, to terminate a program, you might be inclined to write code like this:

```
import sys
sys.stderr.write('It failed!\n')
raise SystemExit(1)
```

None of the extra steps involving `import` or writing to `sys.stderr` are necessary if you simply supply the message to `SystemExit()` instead.

## 13.3. Parsing Command-Line Options

### Problem

You want to write a program that parses options supplied on the command line (found in `sys.argv`).

### Solution

The `argparse` module can be used to parse command-line options. A simple example will help to illustrate the essential features:

```
# search.py
'''
Hypothetical command-line tool for searching a collection of
files for one or more text patterns.
'''

import argparse
parser = argparse.ArgumentParser(description='Search some files')

parser.add_argument(dest='filenames', metavar='filename', nargs='*')

parser.add_argument('-p', '--pat', metavar='pattern', required=True,
                    dest='patterns', action='append',
                    help='text pattern to search for')

parser.add_argument('-v', dest='verbose', action='store_true',
                    help='verbose mode')

parser.add_argument('-o', dest='outfile', action='store',
                    help='output file')

parser.add_argument('--speed', dest='speed', action='store',
                    choices=['slow', 'fast'], default='slow',
                    help='search speed')

args = parser.parse_args()
```

```
# Output the collected arguments
print(args filenames)
print(args patterns)
print(args verbose)
print(args outfile)
print(args speed)
```

This program defines a command-line parser with the following usage:

```
bash % python3 search.py -h
usage: search.py [-h] [-p pattern] [-v] [-o OUTFILE] [--speed {slow,fast}]
               [filename [filename ...]]
```

Search some files

positional arguments:  
filename

optional arguments:  
-h, --help show this help message and exit  
-p pattern, --pat pattern text pattern to search for  
-v verbose mode  
-o OUTFILE output file  
--speed {slow,fast} search speed

The following session shows how data shows up in the program. Carefully observe the output of the `print()` statements.

```
bash % python3 search.py foo.txt bar.txt
usage: search.py [-h] -p pattern [-v] [-o OUTFILE] [--speed {fast,slow}]
               [filename [filename ...]]
search.py: error: the following arguments are required: -p/--pat

bash % python3 search.py -v -p spam --pat=eggs foo.txt bar.txt
filenames = ['foo.txt', 'bar.txt']
patterns   = ['spam', 'eggs']
verbose    = True
outfile    = None
speed      = slow

bash % python3 search.py -v -p spam --pat=eggs foo.txt bar.txt -o results
filenames = ['foo.txt', 'bar.txt']
patterns   = ['spam', 'eggs']
verbose    = True
outfile    = results
speed      = slow

bash % python3 search.py -v -p spam --pat=eggs foo.txt bar.txt -o results \
               --speed=fast
filenames = ['foo.txt', 'bar.txt']
patterns   = ['spam', 'eggs']
verbose    = True
outfile    = results
speed      = fast
```



Further processing of the options is up to the program. Replace the `print()` functions with something more interesting.

## Discussion

The `argparse` module is one of the largest modules in the standard library, and has a huge number of configuration options. This recipe shows an essential subset that can be used and extended to get started.

To parse options, you first create an `ArgumentParser` instance and add declarations for the options you want to support it using the `add_argument()` method. In each `add_argument()` call, the `dest` argument specifies the name of an attribute where the result of parsing will be placed. The `metavar` argument is used when generating help messages. The `action` argument specifies the processing associated with the argument and is often `store` for storing a value or `append` for collecting multiple argument values into a list.

The following argument collects all of the extra command-line arguments into a list. It's being used to make a list of filenames in the example:

```
parser.add_argument(dest='filenames', metavar='filename', nargs='*')
```

The following argument sets a Boolean flag depending on whether or not the argument was provided:

```
parser.add_argument('-v', dest='verbose', action='store_true',
                    help='verbose mode')
```

The following argument takes a single value and stores it as a string:

```
parser.add_argument('-o', dest='outfile', action='store',
                    help='output file')
```

The following argument specification allows an argument to be repeated multiple times and all of the values append into a list. The `required` flag means that the argument must be supplied at least once. The use of `-p` and `--pat` mean that either argument name is acceptable.

```
parser.add_argument('-p', '--pat', metavar='pattern', required=True,
                    dest='patterns', action='append',
                    help='text pattern to search for')
```

Finally, the following argument specification takes a value, but checks it against a set of possible choices.

```
parser.add_argument('--speed', dest='speed', action='store',
                    choices=['slow', 'fast'], default='slow',
                    help='search speed')
```

Once the options have been given, you simply execute the `parser.parse()` method. This will process the `sys.argv` value and return an instance with the results. The results

for each argument are placed into an attribute with the name given in the `dest` parameter to `add_argument()`.

There are several other approaches for parsing command-line options. For example, you might be inclined to manually process `sys.argv` yourself or use the `getopt` module (which is modeled after a similarly named C library). However, if you take this approach, you'll simply end up replicating much of the code that `argparse` already provides. You may also encounter code that uses the `optparse` library to parse options. Although `optparse` is very similar to `argparse`, the latter is more modern and should be preferred in new projects.

## 13.4. Prompting for a Password at Runtime

### Problem

You've written a script that requires a password, but since the script is meant for interactive use, you'd like to prompt the user for a password rather than hardcode it into the script.

### Solution

Python's `getpass` module is precisely what you need in this situation. It will allow you to very easily prompt for a password without having the keyed-in password displayed on the user's terminal. Here's how it's done:

```
import getpass

user = getpass.getuser()
passwd = getpass.getpass()

if svc_login(user, passwd):    # You must write svc_login()
    print('Yay!')
else:
    print('Boo!')
```

In this code, the `svc_login()` function is code that you must write to further process the password entry. Obviously, the exact handling is application-specific.

### Discussion

Note in the preceding code that `getpass.getuser()` doesn't prompt the user for their username. Instead, it uses the current user's login name, according to the user's shell environment, or as a last resort, according to the local system's password database (on platforms that support the `pwd` module).

If you want to explicitly prompt the user for their username, which can be more reliable, use the built-in `input` function:

```
user = input('Enter your username: ')
```

It's also important to remember that some systems may not support the hiding of the typed password input to the `getpass()` method. In this case, Python does all it can to forewarn you of problems (i.e., it alerts you that passwords will be shown in cleartext) before moving on.

## 13.5. Getting the Terminal Size

### Problem

You need to get the terminal size in order to properly format the output of your program.

### Solution

Use the `os.get_terminal_size()` function to do this:

```
>>> import os
>>> sz = os.get_terminal_size()
>>> sz
os.terminal_size(columns=80, lines=24)
>>> sz.columns
80
>>> sz.lines
24
>>>
```

### Discussion

There are many other possible approaches for obtaining the terminal size, ranging from reading environment variables to executing low-level system calls involving `ioctl()` and TTYs. Frankly, why would you bother with that when this one simple call will suffice?

## 13.6. Executing an External Command and Getting Its Output

### Problem

You want to execute an external command and collect its output as a Python string.

## Solution

Use the `subprocess.check_output()` function. For example:

```
import subprocess
out_bytes = subprocess.check_output(['netstat', '-a'])
```

This runs the specified command and returns its output as a byte string. If you need to interpret the resulting bytes as text, add a further decoding step. For example:

```
out_text = out_bytes.decode('utf-8')
```

If the executed command returns a nonzero exit code, an exception is raised. Here is an example of catching errors and getting the output created along with the exit code:

```
try:
    out_bytes = subprocess.check_output(['cmd', 'arg1', 'arg2'])
except subprocess.CalledProcessError as e:
    out_bytes = e.output      # Output generated before error
    code      = e.returncode  # Return code
```

By default, `check_output()` only returns output written to standard output. If you want both standard output and error collected, use the `stderr` argument:

```
out_bytes = subprocess.check_output(['cmd', 'arg1', 'arg2'],
                                     stderr=subprocess.STDOUT)
```

If you need to execute a command with a timeout, use the `timeout` argument:

```
try:
    out_bytes = subprocess.check_output(['cmd', 'arg1', 'arg2'], timeout=5)
except subprocess.TimeoutExpired as e:
    ...
```

Normally, commands are executed without the assistance of an underlying shell (e.g., `sh`, `bash`, etc.). Instead, the list of strings supplied are given to a low-level system command, such as `os.execve()`. If you want the command to be interpreted by a shell, supply it using a simple string and give the `shell=True` argument. This is sometimes useful if you're trying to get Python to execute a complicated shell command involving pipes, I/O redirection, and other features. For example:

```
out_bytes = subprocess.check_output('grep python | wc > out', shell=True)
```

Be aware that executing commands under the shell is a potential security risk if arguments are derived from user input. The `shlex.quote()` function can be used to properly quote arguments for inclusion in shell commands in this case.

## Discussion

The `check_output()` function is the easiest way to execute an external command and get its output. However, if you need to perform more advanced communication with a

subprocess, such as sending it input, you'll need to take a different approach. For that, use the `subprocess.Popen` class directly. For example:

```
import subprocess

# Some text to send
text = b'''
hello world
this is a test
goodbye
'''

# Launch a command with pipes
p = subprocess.Popen(['wc'],
                      stdout = subprocess.PIPE,
                      stdin = subprocess.PIPE)

# Send the data and get the output
stdout, stderr = p.communicate(text)

# To interpret as text, decode
out = stdout.decode('utf-8')
err = stderr.decode('utf-8')
```

The `subprocess` module is not suitable for communicating with external commands that expect to interact with a proper TTY. For example, you can't use it to automate tasks that ask the user to enter a password (e.g., a `ssh` session). For that, you would need to turn to a third-party module, such as those based on the popular “expect” family of tools (e.g., `pexpect` or similar).

## 13.7. Copying or Moving Files and Directories

### Problem

You need to copy or move files and directories around, but you don't want to do it by calling out to shell commands.

### Solution

The `shutil` module has portable implementations of functions for copying files and directories. The usage is extremely straightforward. For example:

```
import shutil

# Copy src to dst. (cp src dst)
shutil.copy(src, dst)

# Copy files, but preserve metadata (cp -p src dst)
shutil.copy2(src, dst)
```

```
# Copy directory tree (cp -R src dst)
shutil.copytree(src, dst)

# Move src to dst (mv src dst)
shutil.move(src, dst)
```

The arguments to these functions are all strings supplying file or directory names. The underlying semantics try to emulate that of similar Unix commands, as shown in the comments.

By default, symbolic links are followed by these commands. For example, if the source file is a symbolic link, then the destination file will be a copy of the file the link points to. If you want to copy the symbolic link instead, supply the `follow_symlinks` keyword argument like this:

```
shutil.copy2(src, dst, follow_symlinks=False)
```

If you want to preserve symbolic links in copied directories, do this:

```
shutil.copytree(src, dst, symlinks=True)
```

The `copytree()` optionally allows you to ignore certain files and directories during the copy process. To do this, you supply an `ignore` function that takes a directory name and filename listing as input, and returns a list of names to ignore as a result. For example:

```
def ignore_pyc_files(dirname, filenames):
    return [name in filenames if name.endswith('.pyc')]

shutil.copytree(src, dst, ignore=ignore_pyc_files)
```

Since ignoring filename patterns is common, a utility function `ignore_patterns()` has already been provided to do it. For example:

```
shutil.copytree(src, dst, ignore=shutil.ignore_patterns('*~', '*.pyc'))
```

## Discussion

Using `shutil` to copy files and directories is mostly straightforward. However, one caution concerning file metadata is that functions such as `copy2()` only make a best effort in preserving this data. Basic information, such as access times, creation times, and permissions, will always be preserved, but preservation of owners, ACLs, resource forks, and other extended file metadata may or may not work depending on the underlying operating system and the user's own access permissions. You probably wouldn't want to use a function like `shutil.copytree()` to perform system backups.

When working with filenames, make sure you use the functions in `os.path` for the greatest portability (especially if working with both Unix and Windows). For example:

```

>>> filename = '/Users/guido/programs/spam.py'
>>> import os.path
>>> os.path.basename(filename)
'spam.py'
>>> os.path.dirname(filename)
'/Users/guido/programs'
>>> os.path.split(filename)
('/Users/guido/programs', 'spam.py')
>>> os.path.join('/new/dir', os.path.basename(filename))
'/new/dir/spam.py'
>>> os.path.expanduser('~ /guido/programs/spam.py')
'/Users/guido/programs/spam.py'
>>>

```

One tricky bit about copying directories with `copytree()` is the handling of errors. For example, in the process of copying, the function might encounter broken symbolic links, files that can't be accessed due to permission problems, and so on. To deal with this, all exceptions encountered are collected into a list and grouped into a single exception that gets raised at the end of the operation. Here is how you would handle it:

```

try:
    shutil.copytree(src, dst)
except shutil.Error as e:
    for src, dst, msg in e.args[0]:
        # src is source name
        # dst is destination name
        # msg is error message from exception
        print(dst, src, msg)

```

If you supply the `ignore_dangling_symlinks=True` keyword argument, then `copytree()` will ignore dangling symlinks.

The functions shown in this recipe are probably the most commonly used. However, `shutil` has many more operations related to copying data. The documentation is definitely worth a further look. See [the Python documentation](#).

## 13.8. Creating and Unpacking Archives

### Problem

You need to create or unpack archives in common formats (e.g., `.tar`, `.tgz`, or `.zip`).

### Solution

The `shutil` module has two functions—`make_archive()` and `unpack_archive()`—that do exactly what you want. For example:

```

>>> import shutil
>>> shutil.unpack_archive('Python-3.3.0.tgz')

```

```
>>> shutil.make_archive('py33', 'zip', 'Python-3.3.0')
'/Users/beazley/Downloads/py33.zip'
>>>
```

The second argument to `make_archive()` is the desired output format. To get a list of supported archive formats, use `get_archive_formats()`. For example:

```
>>> shutil.get_archive_formats()
[('bztar', "bzip2'ed tar-file"), ('gztar', "gzip'ed tar-file"),
 ('tar', 'uncompressed tar file'), ('zip', 'ZIP file')]
>>>
```

## Discussion

Python has other library modules for dealing with the low-level details of various archive formats (e.g., `tarfile`, `zipfile`, `gzip`, `bz2`, etc.). However, if all you’re trying to do is make or extract an archive, there’s really no need to go so low level. You can just use these high-level functions in `shutil` instead.

The functions have a variety of additional options for logging, dryruns, file permissions, and so forth. Consult the `shutil` library documentation for further details.

## 13.9. Finding Files by Name

### Problem

You need to write a script that involves finding files, like a file renaming script or a log archiver utility, but you’d rather not have to call shell utilities from within your Python script, or you want to provide specialized behavior not easily available by “shelling out.”

### Solution

To search for files, use the `os.walk()` function, supplying it with the top-level directory. Here is an example of a function that finds a specific filename and prints out the full path of all matches:

```
#!/usr/bin/env python3.3
import os

def findfile(start, name):
    for relpath, dirs, files in os.walk(start):
        if name in files:
            full_path = os.path.join(start, relpath, name)
            print(os.path.normpath(os.path.abspath(full_path)))

if __name__ == '__main__':
    findfile(sys.argv[1], sys.argv[2])
```



Save this script as *findfile.py* and run it from the command line, feeding in the starting point and the name as positional arguments, like this:

```
bash % ./findfile.py . myfile.txt
```

## Discussion

The `os.walk()` method traverses the directory hierarchy for us, and for each directory it enters, it returns a 3-tuple, containing the relative path to the directory it's inspecting, a list containing all of the directory names in that directory, and a list of filenames in that directory.

For each tuple, you simply check if the target filename is in the `files` list. If it is, `os.path.join()` is used to put together a path. To avoid the possibility of weird looking paths like `./foo//bar`, two additional functions are used to fix the result. The first is `os.path.abspath()`, which takes a path that might be relative and forms the absolute path, and the second is `os.path.normpath()`, which will normalize the path, thereby resolving issues with double slashes, multiple references to the current directory, and so on.

Although this script is pretty simple compared to the features of the `find` utility found on UNIX platforms, it has the benefit of being cross-platform. Furthermore, a lot of additional functionality can be added in a portable manner without much more work. To illustrate, here is a function that prints out all of the files that have a recent modification time:

```
#!/usr/bin/env python3.3

import os
import time

def modified_within(top, seconds):
    now = time.time()
    for path, dirs, files in os.walk(top):
        for name in files:
            fullpath = os.path.join(path, name)
            if os.path.exists(fullpath):
                mtime = os.path.getmtime(fullpath)
                if mtime > (now - seconds):
                    print(fullpath)

if __name__ == '__main__':
    import sys
    if len(sys.argv) != 3:
        print('Usage: {} dir seconds'.format(sys.argv[0]))
        raise SystemExit(1)

    modified_within(sys.argv[1], float(sys.argv[2]))
```

It wouldn't take long for you to build far more complex operations on top of this little function using various features of the `os`, `os.path`, `glob`, and similar modules. See Recipes 5.11 and 5.13 for related recipes.

## 13.10. Reading Configuration Files

### Problem

You want to read configuration files written in the common *.ini* configuration file format.

### Solution

The `configparser` module can be used to read configuration files. For example, suppose you have this configuration file:

```
; config.ini
; Sample configuration file

[installation]
library=%(prefix)s/lib
include=%(prefix)s/include
bin=%(prefix)s/bin
prefix=/usr/local

# Setting related to debug configuration
[debug]
log_errors=true
show_warnings=False

[server]
port: 8080
nworkers: 32
pid-file=/tmp/spam.pid
root=/www/root
signature:
=====
Brought to you by the Python Cookbook
=====
```

Here is an example of how to read it and extract values:

```
>>> from configparser import ConfigParser
>>> cfg = ConfigParser()
>>> cfg.read('config.ini')
['config.ini']
>>> cfg.sections()
['installation', 'debug', 'server']
>>> cfg.get('installation', 'library')
'/usr/local/lib'
>>> cfg.getboolean('debug', 'log_errors')
```

```

True
>>> cfg.getint('server','port')
8080
>>> cfg.getint('server','nworkers')
32
>>> print(cfg.get('server','signature'))

=====
Brought to you by the Python Cookbook
=====
>>>

```

If desired, you can also modify the configuration and write it back to a file using the `cfg.write()` method. For example:

```

>>> cfg.set('server','port','9000')
>>> cfg.set('debug','log_errors','False')
>>> import sys
>>> cfg.write(sys.stdout)
[installation]
library = %(prefix)s/lib
include = %(prefix)s/include
bin = %(prefix)s/bin
prefix = /usr/local

[debug]
log_errors = False
show_warnings = False

[server]
port = 9000
nworkers = 32
pid-file = /tmp/spam.pid
root = /www/root
signature =
=====
Brought to you by the Python Cookbook
=====
>>>

```

## Discussion

Configuration files are well suited as a human-readable format for specifying configuration data to your program. Within each config file, values are grouped into different sections (e.g., “installation,” “debug,” and “server,” in the example). Each section then specifies values for various variables in that section.

There are several notable differences between a config file and using a Python source file for the same purpose. First, the syntax is much more permissive and “sloppy.” For example, both of these assignments are equivalent:

```
prefix=/usr/local
prefix: /usr/local
```

The names used in a config file are also assumed to be case-insensitive. For example:

```
>>> cfg.get('installation', 'PREFIX')
'/usr/local'
>>> cfg.get('installation', 'prefix')
'/usr/local'
>>>
```

When parsing values, methods such as `getboolean()` look for any reasonable value. For example, these are all equivalent:

```
log_errors = true
log_errors = TRUE
log_errors = Yes
log_errors = 1
```

Perhaps the most significant difference between a config file and Python code is that, unlike scripts, configuration files are not executed in a top-down manner. Instead, the file is read in its entirety. If variable substitutions are made, they are done after the fact. For example, in this part of the config file, it doesn't matter that the `prefix` variable is assigned after other variables that happen to use it:

```
[installation]
library=%(prefix)s/lib
include=%(prefix)s/include
bin=%(prefix)s/bin
prefix=/usr/local
```

An easily overlooked feature of `ConfigParser` is that it can read multiple configuration files together and merge their results into a single configuration. For example, suppose a user made their own configuration file that looked like this:

```
; ~/.config.ini
[installation]
prefix=/Users/beazley/test

[debug]
log_errors=False
```

This file can be merged with the previous configuration by reading it separately. For example:

```
>>> # Previously read configuration
>>> cfg.get('installation', 'prefix')
'/usr/local'

>>> # Merge in user-specific configuration
>>> import os
>>> cfg.read(os.path.expanduser('~/.config.ini'))
['/Users/beazley/.config.ini']
```

```
>>> cfg.get('installation', 'prefix')
'/Users/beazley/test'
>>> cfg.get('installation', 'library')
'/Users/beazley/test/lib'
>>> cfg.getboolean('debug', 'log_errors')
False
>>>
```

Observe how the override of the `prefix` variable affects other related variables, such as the setting of `library`. This works because variable interpolation is performed as late as possible. You can see this by trying the following experiment:

```
>>> cfg.get('installation', 'library')
'/Users/beazley/test/lib'
>>> cfg.set('installation', 'prefix', '/tmp/dir')
>>> cfg.get('installation', 'library')
'/tmp/dir/lib'
>>>
```

Finally, it's important to note that Python does not support the full range of features you might find in an `.ini` file used by other programs (e.g., applications on Windows). Make sure you consult the `configparser` documentation for the finer details of the syntax and supported features.

## 13.11. Adding Logging to Simple Scripts

### Problem

You want scripts and simple programs to write diagnostic information to log files.

### Solution

The easiest way to add logging to simple programs is to use the logging module. For example:

```
import logging

def main():
    # Configure the logging system
    logging.basicConfig(
        filename='app.log',
        level=logging.ERROR
    )

    # Variables (to make the calls that follow work)
    hostname = 'www.python.org'
    item = 'spam'
    filename = 'data.csv'
    mode = 'r'
```

```

# Example logging calls (insert into your program)
logging.critical('Host %s unknown', hostname)
logging.error("Couldn't find %r", item)
logging.warning('Feature is deprecated')
logging.info('Opening file %r, mode=%r', filename, mode)
logging.debug('Got here')

if __name__ == '__main__':
    main()

```

The five logging calls (`critical()`, `error()`, `warning()`, `info()`, `debug()`) represent different severity levels in decreasing order. The `level` argument to `basicConfig()` is a filter. All messages issued at a level lower than this setting will be ignored.

The argument to each logging operation is a message string followed by zero or more arguments. When making the final log message, the `%` operator is used to format the message string using the supplied arguments.

If you run this program, the contents of the file *app.log* will be as follows:

```

CRITICAL:root:Host www.python.org unknown
ERROR:root:Could not find 'spam'

```

If you want to change the output or level of output, you can change the parameters to the `basicConfig()` call. For example:

```

logging.basicConfig(
    filename='app.log',
    level=logging.WARNING,
    format='%(levelname)s:%(asctime)s:%(message)s')

```

As a result, the output changes to the following:

```

CRITICAL:2012-11-20 12:27:13,595:Host www.python.org unknown
ERROR:2012-11-20 12:27:13,595:Could not find 'spam'
WARNING:2012-11-20 12:27:13,595:Feature is deprecated

```

As shown, the logging configuration is hardcoded directly into the program. If you want to configure it from a configuration file, change the `basicConfig()` call to the following:

```

import logging
import logging.config

def main():
    # Configure the logging system
    logging.config.fileConfig('logconfig.ini')
    ...

```

Now make a configuration file *logconfig.ini* that looks like this:

```
[loggers]
keys=root

[handlers]
keys=defaultHandler

[formatters]
keys=defaultFormatter

[logger_root]
level=INFO
handlers=defaultHandler
qualname=root

[handler_defaultHandler]
class=FileHandler
formatter=defaultFormatter
args=('app.log', 'a')

[formatter_defaultFormatter]
format=%(levelname)s: %(name)s: %(message)s
```

If you want to make changes to the configuration, you can simply edit the *logconfig.ini* file as appropriate.

## Discussion

Ignoring for the moment that there are about a million advanced configuration options for the logging module, this solution is quite sufficient for simple programs and scripts. Simply make sure that you execute the `basicConfig()` call prior to making any logging calls, and your program will generate logging output.

If you want the logging messages to route to standard error instead of a file, don't supply any filename information to `basicConfig()`. For example, simply do this:

```
logging.basicConfig(level=logging.INFO)
```

One subtle aspect of `basicConfig()` is that it can only be called once in your program. If you later need to change the configuration of the logging module, you need to obtain the root logger and make changes to it directly. For example:

```
logging.getLogger().level = logging.DEBUG
```

It must be emphasized that this recipe only shows a basic use of the logging module. There are significantly more advanced customizations that can be made. An excellent resource for such customization is the “[Logging Cookbook](#)”.

## 13.12. Adding Logging to Libraries

### Problem

You would like to add a logging capability to a library, but don't want it to interfere with programs that don't use logging.

### Solution

For libraries that want to perform logging, you should create a dedicated logger object, and initially configure it as follows:

```
# somelib.py

import logging
log = logging.getLogger(__name__)
log.addHandler(logging.NullHandler())

# Example function (for testing)
def func():
    log.critical('A Critical Error!')
    log.debug('A debug message')
```

With this configuration, no logging will occur by default. For example:

```
>>> import somelib
>>> somelib.func()
>>>
```

However, if the logging system gets configured, log messages will start to appear. For example:

```
>>> import logging
>>> logging.basicConfig()
>>> somelib.func()
CRITICAL:somelib:A Critical Error!
>>>
```

### Discussion

Libraries present a special problem for logging, since information about the environment in which they are used isn't known. As a general rule, you should never write library code that tries to configure the logging system on its own or which makes assumptions about an already existing logging configuration. Thus, you need to take great care to provide isolation.

The call to `getLogger(__name__)` creates a logger module that has the same name as the calling module. Since all modules are unique, this creates a dedicated logger that is likely to be separate from other loggers.



The `log.addHandler(logging.NullHandler())` operation attaches a null handler to the just created logger object. A null handler ignores all logging messages by default. Thus, if the library is used and logging is never configured, no messages or warnings will appear.

One subtle feature of this recipe is that the logging of individual libraries can be independently configured, regardless of other logging settings. For example, consider the following code:

```
>>> import logging
>>> logging.basicConfig(level=logging.ERROR)
>>> import somelib
>>> somelib.func()
CRITICAL:somelib:A Critical Error!

>>> # Change the logging level for 'somelib' only
>>> logging.getLogger('somelib').level=logging.DEBUG
>>> somelib.func()
CRITICAL:somelib:A Critical Error!
DEBUG:somelib:A debug message
>>>
```

Here, the root logger has been configured to only output messages at the ERROR level or higher. However, the level of the logger for `somelib` has been separately configured to output debugging messages. That setting takes precedence over the global setting.

The ability to change the logging settings for a single module like this can be a useful debugging tool, since you don't have to change any of the global logging settings—simply change the level for the one module where you want more output.

The “[Logging HOWTO](#)” has more information about configuring the logging module and other useful tips.

## 13.13. Making a Stopwatch Timer

### Problem

You want to be able to record the time it takes to perform various tasks.

### Solution

The `time` module contains various functions for performing timing-related functions. However, it's often useful to put a higher-level interface on them that mimics a stopwatch. For example:

```

import time

class Timer:
    def __init__(self, func=time.perf_counter):
        self.elapsed = 0.0
        self._func = func
        self._start = None

    def start(self):
        if self._start is not None:
            raise RuntimeError('Already started')
        self._start = self._func()

    def stop(self):
        if self._start is None:
            raise RuntimeError('Not started')
        end = self._func()
        self.elapsed += end - self._start
        self._start = None

    def reset(self):
        self.elapsed = 0.0

    @property
    def running(self):
        return self._start is not None

    def __enter__(self):
        self.start()
        return self

    def __exit__(self, *args):
        self.stop()

```

This class defines a timer that can be started, stopped, and reset as needed by the user. It keeps track of the total elapsed time in the `elapsed` attribute. Here is an example that shows how it can be used:

```

def countdown(n):
    while n > 0:
        n -= 1

# Use 1: Explicit start/stop
t = Timer()
t.start()
countdown(1000000)
t.stop()
print(t.elapsed)

# Use 2: As a context manager
with t:
    countdown(1000000)

```

```
print(t.elapsed)

with Timer() as t2:
    countdown(1000000)
print(t2.elapsed)
```

## Discussion

This recipe provides a simple yet very useful class for making timing measurements and tracking elapsed time. It's also a nice illustration of how to support the context-management protocol and the with statement.

One issue in making timing measurements concerns the underlying time function used to do it. As a general rule, the accuracy of timing measurements made with functions such as `time.time()` or `time.clock()` varies according to the operating system. In contrast, the `time.perf_counter()` function always uses the highest-resolution timer available on the system.

As shown, the time recorded by the `Timer` class is made according to wall-clock time, and includes all time spent sleeping. If you only want the amount of CPU time used by the process, use `time.process_time()` instead. For example:

```
t = Timer(time.process_time)
with t:
    countdown(1000000)
print(t.elapsed)
```

Both the `time.perf_counter()` and `time.process_time()` return a “time” in fractional seconds. However, the actual value of the time doesn't have any particular meaning. To make sense of the results, you have to call the functions twice and compute a time difference.

More examples of timing and profiling are given in [Recipe 14.13](#).

## 13.14. Putting Limits on Memory and CPU Usage

### Problem

You want to place some limits on the memory or CPU use of a program running on Unix system.

### Solution

The resource module can be used to perform both tasks. For example, to restrict CPU time, do the following:

```

import signal
import resource
import os

def time_exceeded(signo, frame):
    print("Time's up!")
    raise SystemExit(1)

def set_max_runtime(seconds):
    # Install the signal handler and set a resource limit
    soft, hard = resource.getrlimit(resource.RLIMIT_CPU)
    resource.setrlimit(resource.RLIMIT_CPU, (seconds, hard))
    signal.signal(signal.SIGXCPU, time_exceeded)

if __name__ == '__main__':
    set_max_runtime(15)
    while True:
        pass

```

When this runs, the SIGXCPU signal is generated when the time expires. The program can then clean up and exit.

To restrict memory use, put a limit on the total address space in use. For example:

```

import resource

def limit_memory(maxsize):
    soft, hard = resource.getrlimit(resource.RLIMIT_AS)
    resource.setrlimit(resource.RLIMIT_AS, (maxsize, hard))

```

With a memory limit in place, programs will start generating `MemoryError` exceptions when no more memory is available.

## Discussion

In this recipe, the `setrlimit()` function is used to set a soft and hard limit on a particular resource. The soft limit is a value upon which the operating system will typically restrict or notify the process via a signal. The hard limit represents an upper bound on the values that may be used for the soft limit. Typically, this is controlled by a system-wide parameter set by the system administrator. Although the hard limit can be lowered, it can never be raised by user processes (even if the process lowered itself).

The `setrlimit()` function can additionally be used to set limits on things such as the number of child processes, number of open files, and similar system resources. Consult the documentation for the `resource` module for further details.

Be aware that this recipe only works on Unix systems, and that it might not work on all of them. For example, when tested, it works on Linux but not on OS X.

## 13.15. Launching a Web Browser

### Problem

You want to launch a browser from a script and have it point to some URL that you specify.

### Solution

The `webbrowser` module can be used to launch a browser in a platform-independent manner. For example:

```
>>> import webbrowser
>>> webbrowser.open('http://www.python.org')
True
>>>
```

This opens the requested page using the default browser. If you want a bit more control over how the page gets opened, you can use one of the following functions:

```
>>> # Open the page in a new browser window
>>> webbrowser.open_new('http://www.python.org')
True
>>>

>>> # Open the page in a new browser tab
>>> webbrowser.open_new_tab('http://www.python.org')
True
>>>
```

These will try to open the page in a new browser window or tab, if possible and supported by the browser.

If you want to open a page in a specific browser, you can use the `webbrowser.get()` function to specify a particular browser. For example:

```
>>> c = webbrowser.get('firefox')
>>> c.open('http://www.python.org')
True
>>> c.open_new_tab('http://docs.python.org')
True
>>>
```

A full list of supported browser names can be found in the [Python documentation](#).

## Discussion

Being able to easily launch a browser can be a useful operation in many scripts. For example, maybe a script performs some kind of deployment to a server and you'd like to have it quickly launch a browser so you can verify that it's working. Or maybe a program writes data out in the form of HTML pages and you'd just like to fire up a browser to see the result. Either way, the `webbrowser` module is a simple solution.

---

# Testing, Debugging, and Exceptions

Testing rocks, but debugging? Not so much. The fact that there's no compiler to analyze your code before Python executes it makes testing a critical part of development. The goal of this chapter is to discuss some common problems related to testing, debugging, and exception handling. It is not meant to be a gentle introduction to test-driven development or the `unittest` module. Thus, some familiarity with testing concepts is assumed.

## 14.1. Testing Output Sent to `stdout`

### Problem

You have a program that has a method whose output goes to standard Output (`sys.stdout`). This almost always means that it emits text to the screen. You'd like to write a test for your code to prove that, given the proper input, the proper output is displayed.

### Solution

Using the `unittest.mock` module's `patch()` function, it's pretty simple to mock out `sys.stdout` for just a single test, and put it back again, without messy temporary variables or leaking mocked-out state between test cases.

Consider, as an example, the following function in a module `mymodule`:

```
# mymodule.py

def urlprint(protocol, host, domain):
    url = '{}://{}.{}'.format(protocol, host, domain)
    print(url)
```

The built-in `print` function, by default, sends output to `sys.stdout`. In order to test that output is actually getting there, you can mock it out using a stand-in object, and then make assertions about what happened. Using the `unittest.mock` module's `patch()` method makes it convenient to replace objects only within the context of a running test, returning things to their original state immediately after the test is complete. Here's the test code for `mymodule`:

```
from io import StringIO
from unittest import TestCase
from unittest.mock import patch
import mymodule

class TestURLPrint(TestCase):
    def test_url_gets_to_stdout(self):
        protocol = 'http'
        host = 'www'
        domain = 'example.com'
        expected_url = '{}://{}.{}\n'.format(protocol, host, domain)

        with patch('sys.stdout', new=StringIO()) as fake_out:
            mymodule.urlprint(protocol, host, domain)
            self.assertEqual(fake_out.getvalue(), expected_url)
```

## Discussion

The `urlprint()` function takes three arguments, and the test starts by setting up dummy arguments for each one. The `expected_url` variable is set to a string containing the expected output.

To run the test, the `unittest.mock.patch()` function is used as a context manager to replace the value of `sys.stdout` with a `StringIO` object as a substitute. The `fake_out` variable is the mock object that's created in this process. This can be used inside the body of the `with` statement to perform various checks. When the `with` statement completes, `patch` conveniently puts everything back the way it was before the test ever ran.

It's worth noting that certain C extensions to Python may write directly to standard output, bypassing the setting of `sys.stdout`. This recipe won't help with that scenario, but it should work fine with pure Python code (if you need to capture I/O from such C extensions, you can do it by opening a temporary file and performing various tricks involving file descriptors to have standard output temporarily redirected to that file).

More information about capturing IO in a string and `StringIO` objects can be found in [Recipe 5.6](#).



## 14.2. Patching Objects in Unit Tests

### Problem

You're writing unit tests and need to apply patches to selected objects in order to make assertions about how they were used in the test (e.g., assertions about being called with certain parameters, access to selected attributes, etc.).

### Solution

The `unittest.mock.patch()` function can be used to help with this problem. It's a little unusual, but `patch()` can be used as a decorator, a context manager, or stand-alone. For example, here's an example of how it's used as a decorator:

```
from unittest.mock import patch
import example

@patch('example.func')
def test1(x, mock_func):
    example.func(x)      # Uses patched example.func
    mock_func.assert_called_with(x)
```

It can also be used as a context manager:

```
with patch('example.func') as mock_func:
    example.func(x)      # Uses patched example.func
    mock_func.assert_called_with(x)
```

Last, but not least, you can use it to patch things manually:

```
p = patch('example.func')
mock_func = p.start()
example.func(x)
mock_func.assert_called_with(x)
p.stop()
```

If necessary, you can stack decorators and context managers to patch multiple objects. For example:

```
@patch('example.func1')
@patch('example.func2')
@patch('example.func3')
def test1(mock1, mock2, mock3):
    ...

def test2():
    with patch('example.patch1') as mock1, \
        patch('example.patch2') as mock2, \
        patch('example.patch3') as mock3:
        ...
```

## Discussion

`patch()` works by taking an existing object with the fully qualified name that you provide and replacing it with a new value. The original value is then restored after the completion of the decorated function or context manager. By default, values are replaced with `MagicMock` instances. For example:

```
>>> x = 42
>>> with patch('__main__.x'):
...     print(x)
...
<MagicMock name='x' id='4314230032'>
>>> x
42
>>>
```

However, you can actually replace the value with anything that you wish by supplying it as a second argument to `patch()`:

```
>>> x
42
>>> with patch('__main__.x', 'patched_value'):
...     print(x)
...
patched_value
>>> x
42
>>>
```

The `MagicMock` instances that are normally used as replacement values are meant to mimic callables and instances. They record information about usage and allow you to make assertions. For example:

```
>>> from unittest.mock import MagicMock
>>> m = MagicMock(return_value = 10)
>>> m(1, 2, debug=True)
10
>>> m.assert_called_with(1, 2, debug=True)
>>> m.assert_called_with(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".../unittest/mock.py", line 726, in assert_called_with
    raise AssertionError(msg)
AssertionError: Expected call: mock(1, 2)
Actual call: mock(1, 2, debug=True)
>>>

>>> m.upper.return_value = 'HELLO'
>>> m.upper('hello')
'HELLO'
>>> assert m.upper.called
```

```

>>> m.split.return_value = ['hello', 'world']
>>> m.split('hello world')
['hello', 'world']
>>> m.split.assert_called_with('hello world')
>>>

>>> m['blah']
<MagicMock name='mock.__getitem__()' id='4314412048'>
>>> m.__getitem__.called
True
>>> m.__getitem__.assert_called_with('blah')
>>>

```

Typically, these kinds of operations are carried out in a unit test. For example, suppose you have some function like this:

```

# example.py
from urllib.request import urlopen
import csv

def dowprices():
    u = urlopen('http://finance.yahoo.com/d/quotes.csv?s=@^DJI&f=s11')
    lines = (line.decode('utf-8') for line in u)
    rows = (row for row in csv.reader(lines) if len(row) == 2)
    prices = { name:float(price) for name, price in rows }
    return prices

```

Normally, this function uses `urlopen()` to go fetch data off the Web and parse it. To unit test it, you might want to give it a more predictable dataset of your own creation, however. Here's an example using patching:

```

import unittest
from unittest.mock import patch
import io
import example

sample_data = io.BytesIO(b'''\
"IBM",91.1\r
"AA",13.25\r
"MSFT",27.72\r
\r
''')

class Tests(unittest.TestCase):
    @patch('example.urlopen', return_value=sample_data)
    def test_dowprices(self, mock_urlopen):
        p = example.dowprices()
        self.assertTrue(mock_urlopen.called)
        self.assertEqual(p,
                         {'IBM': 91.1,
                          'AA': 13.25,
                          'MSFT': 27.72})

```

```
if __name__ == '__main__':
    unittest.main()
```

In this example, the `urlopen()` function in the `example` module is replaced with a mock object that returns a `BytesIO()` containing sample data as a substitute.

An important but subtle facet of this test is the patching of `example.urlopen` instead of `urllib.request.urlopen`. When you are making patches, you have to use the names as they are used in the code being tested. Since the example code uses `from urllib.request import urlopen`, the `urlopen()` function used by the `dowprices()` function is actually located in `example`.

This recipe has really only given a very small taste of what's possible with the `unittest.mock` module. The [official documentation](#) is a must-read for more advanced features.

## 14.3. Testing for Exceptional Conditions in Unit Tests

### Problem

You want to write a unit test that cleanly tests if an exception is raised.

### Solution

To test for exceptions, use the `assertRaises()` method. For example, if you want to test that a function raised a `ValueError` exception, use this code:

```
import unittest

# A simple function to illustrate
def parse_int(s):
    return int(s)

class TestConversion(unittest.TestCase):
    def test_bad_int(self):
        self.assertRaises(ValueError, parse_int, 'N/A')
```

If you need to test the exception's value in some way, then a different approach is needed. For example:

```
import errno

class TestIO(unittest.TestCase):
    def test_file_not_found(self):
        try:
            f = open('/file/not/found')
        except IOError as e:
            self.assertEqual(e.errno, errno.ENOENT)
```

```

else:
    self.fail('IOError not raised')

```

## Discussion

The `assertRaises()` method provides a convenient way to test for the presence of an exception. A common pitfall is to write tests that manually try to do things with exceptions on their own. For instance:

```

class TestConversion(unittest.TestCase):
    def test_bad_int(self):
        try:
            r = parse_int('N/A')
        except ValueError as e:
            self.assertEqual(type(e), ValueError)

```

The problem with such approaches is that it is easy to forget about corner cases, such as that when no exception is raised at all. To do that, you need to add an extra check for that situation, as shown here:

```

class TestConversion(unittest.TestCase):
    def test_bad_int(self):
        try:
            r = parse_int('N/A')
        except ValueError as e:
            self.assertEqual(type(e), ValueError)
        else:
            self.fail('ValueError not raised')

```

The `assertRaises()` method simply takes care of these details, so you should prefer to use it.

The one limitation of `assertRaises()` is that it doesn't provide a means for testing the value of the exception object that's created. To do that, you have to manually test it, as shown. Somewhere in between these two extremes, you might consider using the `assertRaisesRegex()` method, which allows you to test for an exception and perform a regular expression match against the exception's string representation at the same time. For example:

```

class TestConversion(unittest.TestCase):
    def test_bad_int(self):
        self.assertRaisesRegex(ValueError, 'invalid literal .*',
                               parse_int, 'N/A')

```

A little-known fact about `assertRaises()` and `assertRaisesRegex()` is that they can also be used as context managers:

```

class TestConversion(unittest.TestCase):
    def test_bad_int(self):
        with self.assertRaisesRegex(ValueError, 'invalid literal .*'):
            r = parse_int('N/A')

```

This form can be useful if your test involves multiple steps (e.g., setup) besides that of simply executing a callable.

## 14.4. Logging Test Output to a File

### Problem

You want the results of running unit tests written to a file instead of printed to standard output.

### Solution

A very common technique for running unit tests is to include a small code fragment like this at the bottom of your testing file:

```
import unittest

class MyTest(unittest.TestCase):
    ...

if __name__ == '__main__':
    unittest.main()
```

This makes the test file executable, and prints the results of running tests to standard output. If you would like to redirect this output, you need to unwind the `main()` call a bit and write your own `main()` function like this:

```
import sys
def main(out=sys.stderr, verbosity=2):
    loader = unittest.TestLoader()
    suite = loader.loadTestsFromModule(sys.modules[__name__])
    unittest.TextTestRunner(out, verbosity=verbosity).run(suite)

if __name__ == '__main__':
    with open('testing.out', 'w') as f:
        main(f)
```

### Discussion

The interesting thing about this recipe is not so much the task of getting test results redirected to a file, but the fact that doing so exposes some notable inner workings of the `unittest` module.

At a basic level, the `unittest` module works by first assembling a test suite. This test suite consists of the different testing methods you defined. Once the suite has been assembled, the tests it contains are executed.

These two parts of unit testing are separate from each other. The `unittest.TestLoader` instance created in the solution is used to assemble a test suite. The `loadTestsFromModule()` is one of several methods it defines to gather tests. In this case, it scans a module for `TestCase` classes and extracts test methods from them. If you want something more fine-grained, the `loadTestsFromTestCase()` method (not shown) can be used to pull test methods from an individual class that inherits from `TestCase`.

The `TextTestRunner` class is an example of a test runner class. The main purpose of this class is to execute the tests contained in a test suite. This class is the same test runner that sits behind the `unittest.main()` function. However, here we're giving it a bit of low-level configuration, including an output file and an elevated verbosity level.

Although this recipe only consists of a few lines of code, it gives a hint as to how you might further customize the `unittest` framework. To customize how test suites are assembled, you would perform various operations using the `TestLoader` class. To customize how tests execute, you could make custom test runner classes that emulate the functionality of `TextTestRunner`. Both topics are beyond the scope of what can be covered here. However, documentation for the `unittest` module has extensive coverage of the underlying protocols.

## 14.5. Skipping or Anticipating Test Failures

### Problem

You want to skip or mark selected tests as an anticipated failure in your unit tests.

### Solution

The `unittest` module has decorators that can be applied to selected test methods to control their handling. For example:

```
import unittest
import os
import platform

class Tests(unittest.TestCase):
    def test_0(self):
        self.assertTrue(True)

    @unittest.skip('skipped test')
    def test_1(self):
        self.fail('should have failed!')

    @unittest.skipIf(os.name=='posix', 'Not supported on Unix')
    def test_2(self):
        import winreg
```

```

@unittest.skipUnless(platform.system() == 'Darwin', 'Mac specific test')
def test_3(self):
    self.assertTrue(True)

@unittest.expectedFailure
def test_4(self):
    self.assertEqual(2+2, 5)

if __name__ == '__main__':
    unittest.main()

```

If you run this code on a Mac, you'll get this output:

```

bash % python3 testsample.py -v
test_0 (__main__.Tests) ... ok
test_1 (__main__.Tests) ... skipped 'skipped test'
test_2 (__main__.Tests) ... skipped 'Not supported on Unix'
test_3 (__main__.Tests) ... ok
test_4 (__main__.Tests) ... expected failure

-----
Ran 5 tests in 0.002s

OK (skipped=2, expected failures=1)

```

## Discussion

The `skip()` decorator can be used to skip over a test that you don't want to run at all. `skipIf()` and `skipUnless()` can be a useful way to write tests that only apply to certain platforms or Python versions, or which have other dependencies. Use the `@expectedFailure` decorator to mark tests that are known failures, but for which you don't want the test framework to report more information.

The decorators for skipping methods can also be applied to entire testing classes. For example:

```

@unittest.skipUnless(platform.system() == 'Darwin', 'Mac specific tests')
class DarwinTests(unittest.TestCase):
    ...

```

## 14.6. Handling Multiple Exceptions

### Problem

You have a piece of code that can throw any of several different exceptions, and you need to account for all of the potential exceptions that could be raised without creating duplicate code or long, meandering code passages.



## Solution

If you can handle different exceptions all using a single block of code, they can be grouped together in a tuple like this:

```
try:
    client_obj.get_url(url)
except (URLError, ValueError, SocketTimeout):
    client_obj.remove_url(url)
```

In the preceding example, the `remove_url()` method will be called if any one of the listed exceptions occurs. If, on the other hand, you need to handle one of the exceptions differently, put it into its own `except` clause:

```
try:
    client_obj.get_url(url)
except (URLError, ValueError):
    client_obj.remove_url(url)
except SocketTimeout:
    client_obj.handle_url_timeout(url)
```

Many exceptions are grouped into an inheritance hierarchy. For such exceptions, you can catch all of them by simply specifying a base class. For example, instead of writing code like this:

```
try:
    f = open(filename)
except (FileNotFoundError, PermissionError):
    ...
```

you could rewrite the `except` statement as:

```
try:
    f = open(filename)
except OSError:
    ...
```

This works because `OSError` is a base class that's common to both the `FileNotFoundError` and `PermissionError` exceptions.

## Discussion

Although it's not specific to handling *multiple* exceptions per se, it's worth noting that you can get a handle to the thrown exception using the `as` keyword:

```
try:
    f = open(filename)
except OSError as e:
    if e.errno == errno.ENOENT:
        logger.error('File not found')
    elif e.errno == errno.EACCES:
        logger.error('Permission denied')
```

```
else:
    logger.error('Unexpected error: %d', e.errno)
```

In this example, the `e` variable holds an instance of the raised `OSError`. This is useful if you need to inspect the exception further, such as processing it based on the value of an additional status code.

Be aware that `except` clauses are checked in the order listed and that the first match executes. It may be a bit pathological, but you can easily create situations where multiple `except` clauses might match. For example:

```
>>> f = open('missing')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'missing'
>>> try:
...     f = open('missing')
... except OSError:
...     print('It failed')
... except FileNotFoundError:
...     print('File not found')
...
It failed
>>>
```

Here the `except FileNotFoundError` clause doesn't execute because the `OSError` is more general, matches the `FileNotFoundError` exception, and was listed first.

As a debugging tip, if you're not entirely sure about the class hierarchy of a particular exception, you can quickly view it by inspecting the exception's `__mro__` attribute. For example:

```
>>> FileNotFoundError.__mro__
(<class 'FileNotFoundError'>, <class 'OSError'>, <class 'Exception'>,
 <class 'BaseException'>, <class 'object'>)
>>>
```

Any one of the listed classes up to `BaseException` can be used with the `except` statement.

## 14.7. Catching All Exceptions

### Problem

You want to write code that catches all exceptions.

### Solution

To catch all exceptions, write an exception handler for `Exception`, as shown here:

```

try:
    ...
except Exception as e:
    ...
    log('Reason:', e)      # Important!

```

This will catch all exceptions save `SystemExit`, `KeyboardInterrupt`, and `GeneratorExit`. If you also want to catch those exceptions, change `Exception` to `BaseException`.

## Discussion

Catching all exceptions is sometimes used as a crutch by programmers who can't remember all of the possible exceptions that might occur in complicated operations. As such, it is also a very good way to write undebuggable code if you are not careful.

Because of this, if you choose to catch all exceptions, it is absolutely critical to log or report the actual reason for the exception somewhere (e.g., log file, error message printed to screen, etc.). If you don't do this, your head will likely explode at some point. Consider this example:

```

def parse_int(s):
    try:
        n = int(v)
    except Exception:
        print("Couldn't parse")

```

If you try this function, it behaves like this:

```

>>> parse_int('n/a')
Couldn't parse
>>> parse_int('42')
Couldn't parse
>>>

```

At this point, you might be left scratching your head as to why it doesn't work. Now suppose the function had been written like this:

```

def parse_int(s):
    try:
        n = int(v)
    except Exception as e:
        print("Couldn't parse")
        print('Reason:', e)

```

In this case, you get the following output, which indicates that a programming mistake has been made:

```

>>> parse_int('42')
Couldn't parse
Reason: global name 'v' is not defined
>>>

```

All things being equal, it's probably better to be as precise as possible in your exception handling. However, if you must catch all exceptions, just make sure you give good diagnostic information or propagate the exception so that cause doesn't get lost.

## 14.8. Creating Custom Exceptions

### Problem

You're building an application and would like to wrap lower-level exceptions with custom ones that have more meaning in the context of your application.

### Solution

Creating new exceptions is easy—just define them as classes that inherit from `Exception` (or one of the other existing exception types if it makes more sense). For example, if you are writing code related to network programming, you might define some custom exceptions like this:

```
class NetworkError(Exception):
    pass

class HostnameError(NetworkError):
    pass

class TimeoutError(NetworkError):
    pass

class ProtocolError(NetworkError):
    pass
```

Users could then use these exceptions in the normal way. For example:

```
try:
    msg = s.recv()
except TimeoutError as e:
    ...
except ProtocolError as e:
    ...
```

### Discussion

Custom exception classes should almost always inherit from the built-in `Exception` class, or inherit from some locally defined base exception that itself inherits from `Exception`. Although all exceptions also derive from `BaseException`, you should not use this as a base class for new exceptions. `BaseException` is reserved for system-exiting exceptions, such as `KeyboardInterrupt` or `SystemExit`, and other exceptions that should signal the application to exit. Therefore, catching these exceptions is not the

intended use case. Assuming you follow this convention, it follows that inheriting from `BaseException` causes your custom exceptions to not be caught and to signal an imminent application shutdown!

Having custom exceptions in your application and using them as shown makes your application code tell a more coherent story to whoever may need to read the code. One design consideration involves the grouping of custom exceptions via inheritance. In complicated applications, it may make sense to introduce further base classes that group different classes of exceptions together. This gives the user a choice of catching a narrowly specified error, such as this:

```
try:
    s.send(msg)
except ProtocolError:
    ...
```

It also gives the ability to catch a broad range of errors, such as the following:

```
try:
    s.send(msg)
except NetworkError:
    ...
```

If you are going to define a new exception that overrides the `__init__()` method of `Exception`, make sure you always call `Exception.__init__()` with all of the passed arguments. For example:

```
class CustomError(Exception):
    def __init__(self, message, status):
        super().__init__(message, status)
        self.message = message
        self.status = status
```

This might look a little weird, but the default behavior of `Exception` is to accept all arguments passed and to store them in the `.args` attribute as a tuple. Various other libraries and parts of Python expect all exceptions to have the `.args` attribute, so if you skip this step, you might find that your new exception doesn't behave quite right in certain contexts. To illustrate the use of `.args`, consider this interactive session with the built-in `RuntimeError` exception, and notice how any number of arguments can be used with the `raise` statement:

```
>>> try:
...     raise RuntimeError('It failed')
... except RuntimeError as e:
...     print(e.args)
...
('It failed',)
>>> try:
...     raise RuntimeError('It failed', 42, 'spam')
... except RuntimeError as e:
```

```
...     print(e.args)
...
('It failed', 42, 'spam')
>>>
```

For more information on creating your own exceptions, see [the Python documentation](#).

## 14.9. Raising an Exception in Response to Another Exception

### Problem

You want to raise an exception in response to catching a different exception, but want to include information about both exceptions in the traceback.

### Solution

To chain exceptions, use the `raise from` statement instead of a simple `raise` statement. This will give you information about both errors. For example:

```
>>> def example():
...     try:
...         int('N/A')
...     except ValueError as e:
...         raise RuntimeError('A parsing error occurred') from e...
>>>
example()
Traceback (most recent call last):
  File "<stdin>", line 3, in example
ValueError: invalid literal for int() with base 10: 'N/A'
```

The above exception was the direct cause of the following exception:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in example
RuntimeError: A parsing error occurred
>>>
```

As you can see in the traceback, both exceptions are captured. To catch such an exception, you would use a normal `except` statement. However, you can look at the `__cause__` attribute of the exception object to follow the exception chain should you wish. For example:

```
try:
    example()
except RuntimeError as e:
    print("It didn't work:", e)
```

```

if e.__cause__:
    print('Cause:', e.__cause__)

```

An implicit form of chained exceptions occurs when another exception gets raised inside an except block. For example:

```

>>> def example2():
...     try:
...         int('N/A')
...     except ValueError as e:
...         print("Couldn't parse:", err)
...
>>>
>>> example2()
Traceback (most recent call last):
  File "<stdin>", line 3, in example2
ValueError: invalid literal for int() with base 10: 'N/A'

```

During handling of the above exception, another exception occurred:

```

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in example2
NameError: global name 'err' is not defined
>>>

```

In this example, you get information about both exceptions, but the interpretation is a bit different. In this case, the `NameError` exception is raised as the result of a programming error, not in direct response to the parsing error. For this case, the `__cause__` attribute of an exception is not set. Instead, a `__context__` attribute is set to the prior exception.

If, for some reason, you want to suppress chaining, use `raise from None`:

```

>>> def example3():
...     try:
...         int('N/A')
...     except ValueError:
...         raise RuntimeError('A parsing error occurred') from None...
>>>
>>> example3()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in example3
RuntimeError: A parsing error occurred
>>>

```

## Discussion

In designing code, you should give careful attention to use of the `raise` statement inside of other `except` blocks. In most cases, such `raise` statements should probably be changed to `raise from` statements. That is, you should prefer this style:

```
try:
    ...
except SomeException as e:
    raise DifferentException() from e
```

The reason for doing this is that you are explicitly chaining the causes together. That is, the `DifferentException` is being raised in direct response to getting a `SomeException`. This relationship will be explicitly stated in the resulting traceback.

If you write your code in the following style, you still get a chained exception, but it's often not clear if the exception chain was intentional or the result of an unforeseen programming error:

```
try:
    ...
except SomeException:
    raise DifferentException()
```

When you use `raise from`, you're making it clear that you meant to raise the second exception.

Resist the urge to suppress exception information, as shown in the last example. Although suppressing exception information can lead to smaller tracebacks, it also discards information that might be useful for debugging. All things being equal, it's often best to keep as much information as possible.

## 14.10. Reraising the Last Exception

### Problem

You caught an exception in an `except` block, but now you want to reraise it.

### Solution

Simply use the `raise` statement all by itself. For example:

```
>>> def example():
...     try:
...         int('N/A')
...     except ValueError:
...         print("Didn't work")
...         raise
... 
```



```
>>> example()
Didn't work
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in example
ValueError: invalid literal for int() with base 10: 'N/A'
>>>
```

## Discussion

This problem typically arises when you need to take some kind of action in response to an exception (e.g., logging, cleanup, etc.), but afterward, you simply want to propagate the exception along. A very common use might be in catch-all exception handlers:

```
try:
    ...
except Exception as e:
    # Process exception information in some way
    ...

    # Propagate the exception
    raise
```

## 14.11. Issuing Warning Messages

### Problem

You want to have your program issue warning messages (e.g., about deprecated features or usage problems).

### Solution

To have your program issue a warning message, use the `warnings.warn()` function. For example:

```
import warnings

def func(x, y, logfile=None, debug=False):
    if logfile is not None:
        warnings.warn('logfile argument deprecated', DeprecationWarning)
    ...
```

The arguments to `warn()` are a warning message along with a warning class, which is typically one of the following: `UserWarning`, `DeprecationWarning`, `SyntaxWarning`, `RuntimeWarning`, `ResourceWarning`, or `FutureWarning`.

The handling of warnings depends on how you have executed the interpreter and other configuration. For example, if you run Python with the `-W all` option, you'll get output such as the following:

```
bash % python3 -W all example.py
example.py:5: DeprecationWarning: logfile argument is deprecated
  warnings.warn('logfile argument is deprecated', DeprecationWarning)
```

Normally, warnings just produce output messages on standard error. If you want to turn warnings into exceptions, use the `-W error` option:

```
bash % python3 -W error example.py
Traceback (most recent call last):
  File "example.py", line 10, in <module>
    func(2, 3, logfile='log.txt')
  File "example.py", line 5, in func
    warnings.warn('logfile argument is deprecated', DeprecationWarning)
DeprecationWarning: logfile argument is deprecated
bash %
```

## Discussion

Issuing a warning message is often a useful technique for maintaining software and assisting users with issues that don't necessarily rise to the level of being a full-fledged exception. For example, if you're going to change the behavior of a library or framework, you can start issuing warning messages for the parts that you're going to change while still providing backward compatibility for a time. You can also warn users about problematic usage issues in their code.

As another example of a warning in the built-in library, here is an example of a warning message generated by destroying a file without closing it:

```
>>> import warnings
>>> warnings.simplefilter('always')
>>> f = open('/etc/passwd')
>>> del f
__main__:1: ResourceWarning: unclosed file <_io.TextIOWrapper name='/etc/passwd'
mode='r' encoding='UTF-8'>
>>>
```

By default, not all warning messages appear. The `-W` option to Python can control the output of warning messages. `-W all` will output all warning messages, `-W ignore` ignores all warnings, and `-W error` turns warnings into exceptions. As an alternative, you can use the `warnings.simplefilter()` function to control output, as just shown. An argument of `always` makes all warning messages appear, `ignore` ignores all warnings, and `error` turns warnings into exceptions.

For simple cases, this is all you really need to issue warning messages. The `warnings` module provides a variety of more advanced configuration options related to the filtering and handling of warning messages. See [the Python documentation](#) for more information.

## 14.12. Debugging Basic Program Crashes

### Problem

Your program is broken and you'd like some simple strategies for debugging it.

### Solution

If your program is crashing with an exception, running your program as `python3 -i someprogram.py` can be a useful tool for simply looking around. The `-i` option starts an interactive shell as soon as a program terminates. From there, you can explore the environment. For example, suppose you have this code:

```
# sample.py

def func(n):
    return n + 10

func('Hello')
```

Running `python3 -i` produces the following:

```
bash % python3 -i sample.py
Traceback (most recent call last):
  File "sample.py", line 6, in <module>
    func('Hello')
  File "sample.py", line 4, in func
    return n + 10
TypeError: Can't convert 'int' object to str implicitly
>>> func(10)
20
>>>
```

If you don't see anything obvious, a further step is to launch the Python debugger after a crash. For example:

```
>>> import pdb
>>> pdb.pm()
> sample.py(4)func()
-> return n + 10
(Pdb) w
  sample.py(6)<module>()
-> func('Hello')
> sample.py(4)func()
-> return n + 10
(Pdb) print n
'Hello'
(Pdb) q
>>>
```

If your code is deeply buried in an environment where it is difficult to obtain an interactive shell (e.g., in a server), you can often catch errors and produce tracebacks yourself. For example:

```
import traceback
import sys

try:
    func(arg)
except:
    print('**** AN ERROR OCCURRED ****')
    traceback.print_exc(file=sys.stderr)
```

If your program isn't crashing, but it's producing wrong answers or you're mystified by how it works, there is often nothing wrong with just injecting a few `print()` calls in places of interest. However, if you're going to do that, there are a few related techniques of interest. First, the `traceback.print_stack()` function will create a stack track of your program immediately at that point. For example:

```
>>> def sample(n):
...     if n > 0:
...         sample(n-1)
...     else:
...         traceback.print_stack(file=sys.stderr)
...
>>> sample(5)
File "<stdin>", line 1, in <module>
File "<stdin>", line 3, in sample
File "<stdin>", line 3, in sample
File "<stdin>", line 3, in sample
File "<stdin>", line 3, in sample
File "<stdin>", line 3, in sample
File "<stdin>", line 5, in sample
>>>
```

Alternatively, you can also manually launch the debugger at any point in your program using `pdb.set_trace()` like this:

```
import pdb

def func(arg):
    ...
    pdb.set_trace()
    ...
```

This can be a useful technique for poking around in the internals of a large program and answering questions about the control flow or arguments to functions. For instance, once the debugger starts, you can inspect variables using `print` or type a command such as `w` to get the stack traceback.

## Discussion

Don't make debugging more complicated than it needs to be. Simple errors can often be resolved by merely knowing how to read program tracebacks (e.g., the actual error is usually the last line of the traceback). Inserting a few selected `print()` functions in your code can also work well if you're in the process of developing it and you simply want some diagnostics (just remember to remove the statements later).

A common use of the debugger is to inspect variables inside a function that has crashed. Knowing how to enter the debugger after such a crash has occurred is a useful skill to know.

Inserting statements such as `pdb.set_trace()` can be useful if you're trying to unravel an extremely complicated program where the underlying control flow isn't obvious. Essentially, the program will run until it hits the `set_trace()` call, at which point it will immediately enter the debugger. From there, you can try to make more sense of it.

If you're using an IDE for Python development, the IDE will typically provide its own debugging interface on top of or in place of `pdb`. Consult the manual for your IDE for more information.

## 14.13. Profiling and Timing Your Program

### Problem

You would like to find out where your program spends its time and make timing measurements.

### Solution

If you simply want to time your whole program, it's usually easy enough to use something like the Unix `time` command. For example:

```
bash % time python3 someprogram.py
real 0m13.937s
user 0m12.162s
sys 0m0.098s
bash %
```

On the other extreme, if you want a detailed report showing what your program is doing, you can use the `cProfile` module:

```
bash % python3 -m cProfile someprogram.py
      859647 function calls in 16.016 CPU seconds
```

Ordered by: standard name

ncalls	totttime	percall	cumtime	percall	filename:lineno( <b>function</b> )
263169	0.080	0.000	0.080	0.000	someprogram.py:16(frange)
513	0.001	0.000	0.002	0.000	someprogram.py:30(generate_mandel)
262656	0.194	0.000	15.295	0.000	someprogram.py:32(<genexpr>)
1	0.036	0.036	16.077	16.077	someprogram.py:4(<module>)
262144	15.021	0.000	15.021	0.000	someprogram.py:4(in_mandelbrot)
1	0.000	0.000	0.000	0.000	os.py:746(urandom)
1	0.000	0.000	0.000	0.000	png.py:1056(_readable)
1	0.000	0.000	0.000	0.000	png.py:1073(Reader)
1	0.227	0.227	0.438	0.438	png.py:163(<module>)
512	0.010	0.000	0.010	0.000	png.py:200(group)

...

```
bash %
```

More often than not, profiling your code lies somewhere in between these two extremes. For example, you may already know that your code spends most of its time in a few selected functions. For selected profiling of functions, a short decorator can be useful. For example:

```
# timethis.py

import time
from functools import wraps

def timethis(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.perf_counter()
        r = func(*args, **kwargs)
        end = time.perf_counter()
        print('{0}.{1} : {2}'.format(func.__module__, func.__name__, end - start))
        return r
    return wrapper
```

To use this decorator, you simply place it in front of a function definition to get timings from it. For example:

```
>>> @timethis
... def countdown(n):
...     while n > 0:
...         n -= 1
...
>>> countdown(1000000)
__main__.countdown : 0.803001880645752
>>>
```

To time a block of statements, you can define a context manager. For example:

```

from contextlib import contextmanager

@contextmanager
def timeblock(label):
    start = time.perf_counter()
    try:
        yield
    finally:
        end = time.perf_counter()
        print('{} : {}'.format(label, end - start))

```

Here is an example of how the context manager works:

```

>>> with timeblock('counting'):
...     n = 10000000
...     while n > 0:
...         n -= 1
...
counting : 1.5551159381866455
>>>

```

For studying the performance of small code fragments, the `timeit` module can be useful. For example:

```

>>> from timeit import timeit
>>> timeit('math.sqrt(2)', 'import math')
0.1432319980012835
>>> timeit('sqrt(2)', 'from math import sqrt')
0.10836604500218527
>>>

```

`timeit` works by executing the statement specified in the first argument a million times and measuring the time. The second argument is a setup string that is executed to set up the environment prior to running the test. If you need to change the number of iterations, supply a number argument like this:

```

>>> timeit('math.sqrt(2)', 'import math', number=10000000)
1.434852126003534
>>> timeit('sqrt(2)', 'from math import sqrt', number=10000000)
1.0270336690009572
>>>

```

## Discussion

When making performance measurements, be aware that any results you get are approximations. The `time.perf_counter()` function used in the solution provides the highest-resolution timer possible on a given platform. However, it still measures wall-clock time, and can be impacted by many different factors, such as machine load.

If you are interested in process time as opposed to wall-clock time, use `time.process_time()` instead. For example:

```

from functools import wraps
def timethis(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.process_time()
        r = func(*args, **kwargs)
        end = time.process_time()
        print('{}.{} : {}'.format(func.__module__, func.__name__, end - start))
        return r
    return wrapper

```

Last, but not least, if you’re going to perform detailed timing analysis, make sure to read the documentation for the `time`, `timeit`, and other associated modules, so that you have an understanding of important platform-related differences and other pitfalls.

See [Recipe 13.13](#) for a related recipe on creating a stopwatch timer class.

## 14.14. Making Your Programs Run Faster

### Problem

Your program runs too slow and you’d like to speed it up without the assistance of more extreme solutions, such as C extensions or a just-in-time (JIT) compiler.

### Solution

While the first rule of optimization might be to “not do it,” the second rule is almost certainly “don’t optimize the unimportant.” To that end, if your program is running slow, you might start by profiling your code as discussed in [Recipe 14.13](#).

More often than not, you’ll find that your program spends its time in a few hotspots, such as inner data processing loops. Once you’ve identified those locations, you can use the no-nonsense techniques presented in the following sections to make your program run faster.

#### Use functions

A lot of programmers start using Python as a language for writing simple scripts. When writing scripts, it is easy to fall into a practice of simply writing code with very little structure. For example:

```

# somescript.py

import sys
import csv

with open(sys.argv[1]) as f:
    for row in csv.reader(f):

```



```
# Some kind of processing
...
```

A little-known fact is that code defined in the global scope like this runs slower than code defined in a function. The speed difference has to do with the implementation of local versus global variables (operations involving locals are faster). So, if you want to make the program run faster, simply put the scripting statements in a function:

```
# somescript.py
import sys
import csv

def main(filename):
    with open(filename) as f:
        for row in csv.reader(f):
            # Some kind of processing
        ...

main(sys.argv[1])
```

The speed difference depends heavily on the processing being performed, but in our experience, speedups of 15-30% are not uncommon.

### Selectively eliminate attribute access

Every use of the dot (.) operator to access attributes comes with a cost. Under the covers, this triggers special methods, such as `__getattribute__()` and `__getattr__()`, which often lead to dictionary lookups.

You can often avoid attribute lookups by using the `from module import name` form of import as well as making selected use of bound methods. To illustrate, consider the following code fragment:

```
import math

def compute_roots(nums):
    result = []
    for n in nums:
        result.append(math.sqrt(n))
    return result

# Test
nums = range(1000000)
for n in range(100):
    r = compute_roots(nums)
```

When tested on our machine, this program runs in about 40 seconds. Now change the `compute_roots()` function as follows:

```
from math import sqrt

def compute_roots(nums):
```

```

result = []
result_append = result.append
for n in nums:
    result_append(sqrt(n))
return result

```

This version runs in about 29 seconds. The only difference between the two versions of code is the elimination of attribute access. Instead of using `math.sqrt()`, the code uses `sqrt()`. The `result.append()` method is additionally placed into a local variable `result_append` and reused in the inner loop.

However, it must be emphasized that these changes only make sense in frequently executed code, such as loops. So, this optimization really only makes sense in carefully selected places.

## Understand locality of variables

As previously noted, local variables are faster than global variables. For frequently accessed names, speedups can be obtained by making those names as local as possible. For example, consider this modified version of the `compute_roots()` function just discussed:

```

import math

def compute_roots(nums):
    sqrt = math.sqrt
    result = []
    result_append = result.append
    for n in nums:
        result_append(sqrt(n))
    return result

```

In this version, `sqrt` has been lifted from the `math` module and placed into a local variable. If you run this code, it now runs in about 25 seconds (an improvement over the previous version, which took 29 seconds). That additional speedup is due to a local lookup of `sqrt` being a bit faster than a global lookup of `sqrt`.

Locality arguments also apply when working in classes. In general, looking up a value such as `self.name` will be considerably slower than accessing a local variable. In inner loops, it might pay to lift commonly accessed attributes into a local variable. For example:

```

# Slower
class SomeClass:
    ...
    def method(self):
        for x in s:
            op(self.value)

# Faster
class SomeClass:

```

```
...
def method(self):
    value = self.value
    for x in s:
        op(value)
```

## Avoid gratuitous abstraction

Any time you wrap up code with extra layers of processing, such as decorators, properties, or descriptors, you're going to make it slower. As an example, consider this class:

```
class A:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    @property
    def y(self):
        return self._y
    @y.setter
    def y(self, value):
        self._y = value
```

Now, try a simple timing test:

```
>>> from timeit import timeit
>>> a = A(1,2)
>>> timeit('a.x', 'from __main__ import a')
0.07817923510447145
>>> timeit('a.y', 'from __main__ import a')
0.35766440676525235
>>>
```

As you can observe, accessing the property `y` is not just slightly slower than a simple attribute `x`, it's about 4.5 times slower. If this difference matters, you should ask yourself if the definition of `y` as a property was really necessary. If not, simply get rid of it and go back to using a simple attribute instead. Just because it might be common for programs in another programming language to use getter/setter functions, that doesn't mean you should adopt that programming style for Python.

## Use the built-in containers

Built-in data types such as strings, tuples, lists, sets, and dicts are all implemented in C, and are rather fast. If you're inclined to make your own data structures as a replacement (e.g., linked lists, balanced trees, etc.), it may be rather difficult if not impossible to match the speed of the built-ins. Thus, you're often better off just using them.

## Avoid making unnecessary data structures or copies

Sometimes programmers get carried away with making unnecessary data structures when they just don't have to. For example, someone might write code like this:

```
values = [x for x in sequence]
squares = [x*x for x in values]
```

Perhaps the thinking here is to first collect a bunch of values into a list and then to start applying operations such as list comprehensions to it. However, the first list is completely unnecessary. Simply write the code like this:

```
squares = [x*x for x in sequence]
```

Related to this, be on the lookout for code written by programmers who are overly paranoid about Python's sharing of values. Overuse of functions such as `copy.deepcopy()` may be a sign of code that's been written by someone who doesn't fully understand or trust Python's memory model. In such code, it may be safe to eliminate many of the copies.

## Discussion

Before optimizing, it's usually worthwhile to study the algorithms that you're using first. You'll get a much bigger speedup by switching to an  $O(n \log n)$  algorithm than by trying to tweak the implementation of an  $O(n^2)$  algorithm.

If you've decided that you still must optimize, it pays to consider the big picture. As a general rule, you don't want to apply optimizations to every part of your program, because such changes are going to make the code hard to read and understand. Instead, focus only on known performance bottlenecks, such as inner loops.

You need to be especially wary interpreting the results of micro-optimizations. For example, consider these two techniques for creating a dictionary:

```
a = {
    'name' : 'AAPL',
    'shares' : 100,
    'price' : 534.22
}

b = dict(name='AAPL', shares=100, price=534.22)
```

The latter choice has the benefit of less typing (you don't need to quote the key names). However, if you put the two code fragments in a head-to-head performance battle, you'll find that using `dict()` runs three times slower! With this knowledge, you might be inclined to scan your code and replace every use of `dict()` with its more verbose alternative. However, a smart programmer will only focus on parts of a program where it might actually matter, such as an inner loop. In other places, the speed difference just isn't going to matter at all.

If, on the other hand, your performance needs go far beyond the simple techniques in this recipe, you might investigate the use of tools based on just-in-time (JIT) compilation techniques. For example, the [PyPy project](#) is an alternate implementation of the Python

interpreter that analyzes the execution of your program and generates native machine code for frequently executed parts. It can sometimes make Python programs run an order of magnitude faster, often approaching (or even exceeding) the speed of code written in C. Unfortunately, as of this writing, PyPy does not yet fully support Python 3. So, that is something to look for in the future. You might also consider the [Numba project](#). Numba is a dynamic compiler where you annotate selected Python functions that you want to optimize with a decorator. Those functions are then compiled into native machine code through the use of [LLVM](#). It too can produce significant performance gains. However, like PyPy, support for Python 3 should be viewed as somewhat experimental.

Last, but not least, the words of John Ousterhout come to mind: “The best performance improvement is the transition from the nonworking to the working state.” Don’t worry about optimization until you need to. Making sure your program works correctly is usually more important than making it run fast (at least initially).



---

## CHAPTER 15

# C Extensions

This chapter looks at the problem of accessing C code from Python. Many of Python's built-in libraries are written in C, and accessing C is an important part of making Python talk to existing libraries. It's also an area that might require the most study if you're faced with the problem of porting extension code from Python 2 to 3.

Although Python provides an extensive C programming API, there are actually many different approaches for dealing with C. Rather than trying to give an exhaustive reference for every possible tool or technique, the approach is to focus on a small fragment of C code along with some representative examples of how to work with the code. The goal is to provide a series of programming templates that experienced programmers can expand upon for their own use.

Here is the C code we will work with in most of the recipes:

```
/* sample.c */_method
#include <math.h>

/* Compute the greatest common divisor */
int gcd(int x, int y) {
    int g = y;
    while (x > 0) {
        g = x;
        x = y % x;
        y = g;
    }
    return g;
}

/* Test if (x0,y0) is in the Mandelbrot set or not */
int in_mandel(double x0, double y0, int n) {
    double x=0,y=0,xtemp;
    while (n > 0) {
        xtemp = x*x - y*y + x0;
        y = 2*x*y + y0;
    }
}
```

```

    x = xtemp;
    n -= 1;
    if (x*x + y*y > 4) return 0;
}
return 1;
}

/* Divide two numbers */
int divide(int a, int b, int *remainder) {
    int quot = a / b;
    *remainder = a % b;
    return quot;
}

/* Average values in an array */
double avg(double *a, int n) {
    int i;
    double total = 0.0;
    for (i = 0; i < n; i++) {
        total += a[i];
    }
    return total / n;
}

/* A C data structure */
typedef struct Point {
    double x,y;
} Point;

/* Function involving a C data structure */
double distance(Point *p1, Point *p2) {
    return hypot(p1->x - p2->x, p1->y - p2->y);
}

```

This code contains a number of different C programming features. First, there are a few simple functions such as `gcd()` and `is_mandel()`. The `divide()` function is an example of a C function returning multiple values, one through a pointer argument. The `avg()` function performs a data reduction across a C array. The `Point` and `distance()` function involve C structures.

For all of the recipes that follow, assume that the preceding code is found in a file named *sample.c*, that definitions are found in a file named *sample.h* and that it has been compiled into a library `libsample` that can be linked to other C code. The exact details of compilation and linking vary from system to system, but that is not the primary focus. It is assumed that if you're working with C code, you've already figured that out.



## 15.1. Accessing C Code Using ctypes

### Problem

You have a small number of C functions that have been compiled into a shared library or DLL. You would like to call these functions purely from Python without having to write additional C code or using a third-party extension tool.

### Solution

For small problems involving C code, it is often easy enough to use the `ctypes` module that is part of Python's standard library. In order to use `ctypes`, you must first make sure the C code you want to access has been compiled into a shared library that is compatible with the Python interpreter (e.g., same architecture, word size, compiler, etc.). For the purposes of this recipe, assume that a shared library, `libsampl.e.so`, has been created and that it contains nothing more than the code shown in the chapter introduction. Further assume that the `libsampl.e.so` file has been placed in the same directory as the `sample.py` file shown next.

To access the resulting library, you make a Python module that wraps around it, such as the following:

```
# sample.py
import ctypes
import os

# Try to locate the .so file in the same directory as this file
_file = 'libsampl.e.so'
_path = os.path.join(*(os.path.split(__file__)[:-1] + (_file,)))
_mod = ctypes.cdll.LoadLibrary(_path)

# int gcd(int, int)
gcd = _mod.gcd
gcd.argtypes = (ctypes.c_int, ctypes.c_int)
gcd.restype = ctypes.c_int

# int in_mandel(double, double, int)
in_mandel = _mod.in_mandel
in_mandel.argtypes = (ctypes.c_double, ctypes.c_double, ctypes.c_int)
in_mandel.restype = ctypes.c_int

# int divide(int, int, int *)
_divide = _mod.divide
_divide.argtypes = (ctypes.c_int, ctypes.c_int, ctypes.POINTER(ctypes.c_int))
_divide.restype = ctypes.c_int

def divide(x, y):
    rem = ctypes.c_int()
    quot = _divide(x, y, rem)
```

```

        return quot,rem.value

# void avg(double *, int n)
# Define a special type for the 'double *' argument
class DoubleArrayType:
    def from_param(self, param):
        typename = type(param).__name__
        if hasattr(self, 'from_' + typename):
            return getattr(self, 'from_' + typename)(param)
        elif isinstance(param, ctypes.Array):
            return param
        else:
            raise TypeError("Can't convert %s" % typename)

    # Cast from array.array objects
    def from_array(self, param):
        if param.typecode != 'd':
            raise TypeError('must be an array of doubles')
        ptr, _ = param.buffer_info()
        return ctypes.cast(ptr, ctypes.POINTER(ctypes.c_double))

    # Cast from lists/tuples
    def from_list(self, param):
        val = ((ctypes.c_double)*len(param))(*param)
        return val

    from_tuple = from_list

    # Cast from a numpy array
    def from_ndarray(self, param):
        return param.ctypes.data_as(ctypes.POINTER(ctypes.c_double))

DoubleArray = DoubleArrayType()
_avg = _mod.avg
_avg.argtypes = (DoubleArray, ctypes.c_int)
_avg.restype = ctypes.c_double

def avg(values):
    return _avg(values, len(values))

# struct Point { }
class Point(ctypes.Structure):
    _fields_ = [('x', ctypes.c_double),
                ('y', ctypes.c_double)]

# double distance(Point *, Point *)
distance = _mod.distance
distance.argtypes = (ctypes.POINTER(Point), ctypes.POINTER(Point))
distance.restype = ctypes.c_double

```

If all goes well, you should be able to load the module and use the resulting C functions. For example:

```

>>> import sample
>>> sample.gcd(35,42)
7
>>> sample.in_mandel(0,0,500)
1
>>> sample.in_mandel(2.0,1.0,500)
0
>>> sample.divide(42,8)
(5, 2)
>>> sample.avg([1,2,3])
2.0
>>> p1 = sample.Point(1,2)
>>> p2 = sample.Point(4,5)
>>> sample.distance(p1,p2)
4.242640687119285
>>>

```

## Discussion

There are several aspects of this recipe that warrant some discussion. The first issue concerns the overall packaging of C and Python code together. If you are using `ctypes` to access C code that you have compiled yourself, you will need to make sure that the shared library gets placed in a location where the `sample.py` module can find it. One possibility is to put the resulting `.so` file in the same directory as the supporting Python code. This is what's shown at the first part of this recipe—`sample.py` looks at the `__file__` variable to see where it has been installed, and then constructs a path that points to a `libsampl.so` file in the same directory.

If the C library is going to be installed elsewhere, then you'll have to adjust the path accordingly. If the C library is installed as a standard library on your machine, you might be able to use the `ctypes.util.find_library()` function. For example:

```

>>> from ctypes.util import find_library
>>> find_library('m')
'/usr/lib/libm.dylib'
>>> find_library('pthread')
'/usr/lib/libpthread.dylib'
>>> find_library('sample')
'/usr/local/lib/libsample.so'
>>>

```

Again, `ctypes` won't work at all if it can't locate the library with the C code. Thus, you'll need to spend a few minutes thinking about how you want to install things.

Once you know where the C library is located, you use `ctypes.cdll.LoadLibrary()` to load it. The following statement in the solution does this where `_path` is the full pathname to the shared library:

```

_mod = ctypes.cdll.LoadLibrary(_path)

```

Once a library has been loaded, you need to write statements that extract specific symbols and put type signatures on them. This is what's happening in code fragments such as this:

```
# int in_mandel(double, double, int)
in_mandel = _mod.in_mandel
in_mandel.argtypes = (ctypes.c_double, ctypes.c_double, ctypes.c_int)
in_mandel.restype = ctypes.c_int
```

In this code, the `.argtypes` attribute is a tuple containing the input arguments to a function, and `.restype` is the return type. `ctypes` defines a variety of type objects (e.g., `c_double`, `c_int`, `c_short`, `c_float`, etc.) that represent common C data types. Attaching the type signatures is critical if you want to make Python pass the right kinds of arguments and convert data correctly (if you don't do this, not only will the code not work, but you might cause the entire interpreter process to crash).

A somewhat tricky part of using `ctypes` is that the original C code may use idioms that don't map cleanly to Python. The `divide()` function is a good example because it returns a value through one of its arguments. Although that's a common C technique, it's often not clear how it's supposed to work in Python. For example, you can't do anything straightforward like this:

```
>>> divide = _mod.divide
>>> divide.argtypes = (ctypes.c_int, ctypes.c_int, ctypes.POINTER(ctypes.c_int))
>>> x = 0
>>> divide(10, 3, x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 3: <class 'TypeError'>: expected LP_c_int
instance instead of int
>>>
```

Even if this did work, it would violate Python's immutability of integers and probably cause the entire interpreter to be sucked into a black hole. For arguments involving pointers, you usually have to construct a compatible `ctypes` object and pass it in like this:

```
>>> x = ctypes.c_int()
>>> divide(10, 3, x)
3
>>> x.value
1
>>>
```

Here an instance of a `ctypes.c_int` is created and passed in as the pointer object. Unlike a normal Python integer, a `c_int` object can be mutated. The `.value` attribute can be used to either retrieve or change the value as desired.

For cases where the C calling convention is “un-Pythonic,” it is common to write a small wrapper function. In the solution, this code makes the `divide()` function return the two results using a tuple instead:

```
# int divide(int, int, int *)
_divide = _mod.divide
_divide.argtypes = (ctypes.c_int, ctypes.c_int, ctypes.POINTER(ctypes.c_int))
_divide.restype = ctypes.c_int

def divide(x, y):
    rem = ctypes.c_int()
    quot = _divide(x,y,rem)
    return quot, rem.value
```

The `avg()` function presents a new kind of challenge. The underlying C code expects to receive a pointer and a length representing an array. However, from the Python side, we must consider the following questions: What is an array? Is it a list? A tuple? An array from the `array` module? A `numpy` array? Is it all of these? In practice, a Python “array” could take many different forms, and maybe you would like to support multiple possibilities.

The `DoubleArrayType` class shows how to handle this situation. In this class, a single method `from_param()` is defined. The role of this method is to take a single parameter and narrow it down to a compatible `ctypes` object (a pointer to a `ctypes.c_double`, in the example). Within `from_param()`, you are free to do anything that you wish. In the solution, the `typename` of the parameter is extracted and used to dispatch to a more specialized method. For example, if a list is passed, the `typename` is `list` and a method `from_list()` is invoked.

For lists and tuples, the `from_list()` method performs a conversion to a `ctypes` array object. This looks a little weird, but here is an interactive example of converting a list to a `ctypes` array:

```
>>> nums = [1, 2, 3]
>>> a = (ctypes.c_double * len(nums))(*nums)
>>> a
<__main__.c_double_Array_3 object at 0x10069cd40>
>>> a[0]
1.0
>>> a[1]
2.0
>>> a[2]
3.0
>>>
```

For array objects, the `from_array()` method extracts the underlying memory pointer and casts it to a `ctypes` pointer object. For example:

```

>>> import array
>>> a = array.array('d',[1,2,3])
>>> a
array('d', [1.0, 2.0, 3.0])
>>> ptr_ = a.buffer_info()
>>> ptr
4298687200
>>> ctypes.cast(ptr, ctypes.POINTER(ctypes.c_double))
<__main__.LP_c_double object at 0x10069cd40>
>>>

```

The `from_ndarray()` shows comparable conversion code for numpy arrays.

By defining the `DoubleArrayType` class and using it in the type signature of `avg()`, as shown, the function can accept a variety of different array-like inputs:

```

>>> import sample
>>> sample.avg([1,2,3])
2.0
>>> sample.avg((1,2,3))
2.0
>>> import array
>>> sample.avg(array.array('d',[1,2,3]))
2.0
>>> import numpy
>>> sample.avg(numpy.array([1.0,2.0,3.0]))
2.0
>>>

```

The last part of this recipe shows how to work with a simple C structure. For structures, you simply define a class that contains the appropriate fields and types like this:

```

class Point(ctypes.Structure):
    _fields_ = [('x', ctypes.c_double),
                ('y', ctypes.c_double)]

```

Once defined, you can use the class in type signatures as well as in code that needs to instantiate and work with the structures. For example:

```

>>> p1 = sample.Point(1,2)
>>> p2 = sample.Point(4,5)
>>> p1.x
1.0
>>> p1.y
2.0
>>> sample.distance(p1,p2)
4.242640687119285
>>>

```

A few final comments: `ctypes` is a useful library to know about if all you're doing is accessing a few C functions from Python. However, if you're trying to access a large library, you might want to look at alternative approaches, such as [Swig](#) (described in [Recipe 15.9](#)) or [Cython](#) (described in [Recipe 15.10](#)).

The main problem with a large library is that since `ctypes` isn't entirely automatic, you'll have to spend a fair bit of time writing out all of the type signatures, as shown in the example. Depending on the complexity of the library, you might also have to write a large number of small wrapper functions and supporting classes. Also, unless you fully understand all of the low-level details of the C interface, including memory management and error handling, it is often quite easy to make Python catastrophically crash with a segmentation fault, access violation, or some similar error.

As an alternative to `ctypes`, you might also look at [CFFI](#). CFFI provides much of the same functionality, but uses C syntax and supports more advanced kinds of C code. As of this writing, CFFI is still a relatively new project, but its use has been growing rapidly. There has even been some discussion of including it in the Python standard library in some future release. Thus, it's definitely something to keep an eye on.

## 15.2. Writing a Simple C Extension Module

### Problem

You want to write a simple C extension module directly using Python's extension API and no other tools.

### Solution

For simple C code, it is straightforward to make a handcrafted extension module. As a preliminary step, you probably want to make sure your C code has a proper header file. For example,

```
/* sample.h */

#include <math.h>

extern int gcd(int, int);
extern int in_mandel(double x0, double y0, int n);
extern int divide(int a, int b, int *remainder);
extern double avg(double *a, int n);

typedef struct Point {
    double x,y;
} Point;

extern double distance(Point *p1, Point *p2);
```

Typically, this header would correspond to a library that has been compiled separately. With that assumption, here is a sample extension module that illustrates the basics of writing extension functions:

```
#include "Python.h"
#include "sample.h"

/* int gcd(int, int) */
static PyObject *py_gcd(PyObject *self, PyObject *args) {
    int x, y, result;

    if (!PyArg_ParseTuple(args, "ii", &x, &y)) {
        return NULL;
    }
    result = gcd(x,y);
    return Py_BuildValue("i", result);
}

/* int in_mandel(double, double, int) */
static PyObject *py_in_mandel(PyObject *self, PyObject *args) {
    double x0, y0;
    int n;
    int result;

    if (!PyArg_ParseTuple(args, "ddi", &x0, &y0, &n)) {
        return NULL;
    }
    result = in_mandel(x0,y0,n);
    return Py_BuildValue("i", result);
}

/* int divide(int, int, int *) */
static PyObject *py_divide(PyObject *self, PyObject *args) {
    int a, b, quotient, remainder;
    if (!PyArg_ParseTuple(args, "ii", &a, &b)) {
        return NULL;
    }
    quotient = divide(a,b, &remainder);
    return Py_BuildValue("(ii)", quotient, remainder);
}

/* Module method table */
static PyMethodDef SampleMethods[] = {
    {"gcd", py_gcd, METH_VARARGS, "Greatest common divisor"},
    {"in_mandel", py_in_mandel, METH_VARARGS, "Mandelbrot test"},
    {"divide", py_divide, METH_VARARGS, "Integer division"},
    { NULL, NULL, 0, NULL}
};

/* Module structure */
static struct PyModuleDef samplemodule = {
    PyModuleDef_HEAD_INIT,
```



```

"sample",          /* name of module */
"A sample module", /* Doc string (may be NULL) */
-1,               /* Size of per-interpreter state or -1 */
SampleMethods     /* Method table */
};

/* Module initialization function */
PyMODINIT_FUNC
PyInit_sample(void) {
    return PyModule_Create(&samplemodule);
}

```

For building the extension module, create a `setup.py` file that looks like this:

```

# setup.py
from distutils.core import setup, Extension

setup(name='sample',
      ext_modules=[
          Extension('sample',
                  ['pysample.c'],
                  include_dirs = ['/some/dir'],
                  define_macros = [('FOO', '1')],
                  undef_macros = ['BAR'],
                  library_dirs = ['/usr/local/lib'],
                  libraries = ['sample']
                  )
      ]
)

```

Now, to build the resulting library, simply use `python3 buildlib.py build_ext --inplace`. For example:

```

bash % python3 setup.py build_ext --inplace
running build_ext
building 'sample' extension
gcc -fno-strict-aliasing -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-prototypes
-I/usr/local/include/python3.3m -c pysample.c
-o build/temp.macosx-10.6-x86_64-3.3/pysample.o
gcc -bundle -undefined dynamic_lookup
build/temp.macosx-10.6-x86_64-3.3/pysample.o \
-L/usr/local/lib -lsample -o sample.so
bash %

```

As shown, this creates a shared library called `sample.so`. When compiled, you should be able to start importing it as a module:

```

>>> import sample
>>> sample.gcd(35, 42)
7
>>> sample.in_mandel(0, 0, 500)
1
>>> sample.in_mandel(2.0, 1.0, 500)

```

```

0
>>> sample.divide(42, 8)
(5, 2)
>>>

```

If you are attempting these steps on Windows, you may need to spend some time fiddling with your environment and the build environment to get extension modules to build correctly. Binary distributions of Python are typically built using Microsoft Visual Studio. To get extensions to work, you may have to compile them using the same or compatible tools. See [the Python documentation](#).

## Discussion

Before attempting any kind of handwritten extension, it is absolutely critical that you consult Python’s documentation on “[Extending and Embedding the Python Interpreter](#)”. Python’s C extension API is large, and repeating all of it here is simply not practical. However, the most important parts can be easily discussed.

First, in extension modules, functions that you write are all typically written with a common prototype such as this:

```

static PyObject *py_func(PyObject *self, PyObject *args) {
    ...
}

```

`PyObject` is the C data type that represents any Python object. At a very high level, an extension function is a C function that receives a tuple of Python objects (in `PyObject *args`) and returns a new Python object as a result. The `self` argument to the function is unused for simple extension functions, but comes into play should you want to define new classes or object types in C (e.g., if the extension function were a method of a class, then `self` would hold the instance).

The `PyArg_ParseTuple()` function is used to convert values from Python to a C representation. As input, it takes a format string that indicates the required values, such as “i” for integer and “d” for double, as well as the addresses of C variables in which to place the converted results. `PyArg_ParseTuple()` performs a variety of checks on the number and type of arguments. If there is any mismatch with the format string, an exception is raised and `NULL` is returned. By checking for this and simply returning `NULL`, an appropriate exception will have been raised in the calling code.

The `Py_BuildValue()` function is used to create Python objects from C data types. It also accepts a format code to indicate the desired type. In the extension functions, it is used to return results back to Python. One feature of `Py_BuildValue()` is that it can build more complicated kinds of objects, such as tuples and dictionaries. In the code for `py_divide()`, an example showing the return of a tuple is shown. However, here are a few more examples:

```

return Py_BuildValue("i", 34);    // Return an integer
return Py_BuildValue("d", 3.4);  // Return a double
return Py_BuildValue("s", "Hello"); // Null-terminated UTF-8 string
return Py_BuildValue("(ii)", 3, 4); // Tuple (3, 4)

```

Near the bottom of any extension module, you will find a function table such as the `SampleMethods` table shown in this recipe. This table lists C functions, the names to use in Python, as well as doc strings. All modules are required to specify such a table, as it gets used in the initialization of the module.

The final function `PyInit_sample()` is the module initialization function that executes when the module is first imported. The primary job of this function is to register the module object with the interpreter.

As a final note, it must be stressed that there is considerably more to extending Python with C functions than what is shown here (in fact, the C API contains well over 500 functions in it). You should view this recipe simply as a stepping stone for getting started. To do more, start with the documentation on the `PyArg_ParseTuple()` and `Py_BuildValue()` functions, and expand from there.

## 15.3. Writing an Extension Function That Operates on Arrays

### Problem

You want to write a C extension function that operates on contiguous arrays of data, as might be created by the array module or libraries like NumPy. However, you would like your function to be general purpose and not specific to any one array library.

### Solution

To receive and process arrays in a portable manner, you should write code that uses the **Buffer Protocol**. Here is an example of a handwritten C extension function that receives array data and calls the `avg(double *buf, int len)` function from this chapter's introduction:

```

/* Call double avg(double *, int) */
static PyObject *py_avg(PyObject *self, PyObject *args) {
    PyObject *bufobj;
    Py_buffer view;
    double result;
    /* Get the passed Python object */
    if (!PyArg_ParseTuple(args, "O", &bufobj)) {
        return NULL;
    }

    /* Attempt to extract buffer information from it */

```

```

if (PyObject_GetBuffer(bufobj, &view,
    PyBUF_ANY_CONTIGUOUS | PyBUF_FORMAT) == -1) {
    return NULL;
}

if (view.ndim != 1) {
    PyErr_SetString(PyExc_TypeError, "Expected a 1-dimensional array");
    PyBuffer_Release(&view);
    return NULL;
}

/* Check the type of items in the array */
if (strcmp(view.format, "d") != 0) {
    PyErr_SetString(PyExc_TypeError, "Expected an array of doubles");
    PyBuffer_Release(&view);
    return NULL;
}

/* Pass the raw buffer and size to the C function */
result = avg(view.buf, view.shape[0]);

/* Indicate we're done working with the buffer */
PyBuffer_Release(&view);
return Py_BuildValue("d", result);
}

```

Here is an example that shows how this extension function works:

```

>>> import array
>>> avg(array.array('d',[1,2,3]))
2.0
>>> import numpy
>>> avg(numpy.array([1.0,2.0,3.0]))
2.0
>>> avg([1,2,3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'list' does not support the buffer interface
>>> avg(b'Hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Expected an array of doubles
>>> a = numpy.array([[1.,2.,3.],[4.,5.,6.]])
>>> avg(a[:,2])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: ndarray is not contiguous
>>> sample.avg(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Expected a 1-dimensional array
>>> sample.avg(a[0])

```

```
2.0
>>>
```

## Discussion

Passing array objects to C functions might be one of the most common things you would want to do with a extension function. A large number of Python applications, ranging from image processing to scientific computing, are based on high-performance array processing. By writing code that can accept and operate on arrays, you can write customized code that plays nicely with those applications as opposed to having some sort of custom solution that only works with your own code.

The key to this code is the `PyBuffer_GetBuffer()` function. Given an arbitrary Python object, it tries to obtain information about the underlying memory representation. If it's not possible, as is the case with most normal Python objects, it simply raises an exception and returns -1. The special flags passed to `PyBuffer_GetBuffer()` give additional hints about the kind of memory buffer that is requested. For example, `PyBUF_ANY_CONTIGUOUS` specifies that a contiguous region of memory is required.

For arrays, byte strings, and other similar objects, a `Py_buffer` structure is filled with information about the underlying memory. This includes a pointer to the memory, size, itemsize, format, and other details. Here is the definition of this structure:

```
typedef struct bufferinfo {
    void *buf;           /* Pointer to buffer memory */
    PyObject *obj;       /* Python object that is the owner */
    Py_ssize_t len;      /* Total size in bytes */
    Py_ssize_t itemsize; /* Size in bytes of a single item */
    int readonly;        /* Read-only access flag */
    int ndim;            /* Number of dimensions */
    char *format;        /* struct code of a single item */
    Py_ssize_t *shape;   /* Array containing dimensions */
    Py_ssize_t *strides; /* Array containing strides */
    Py_ssize_t *suboffsets; /* Array containing suboffsets */
} Py_buffer;
```

In this recipe, we are simply concerned with receiving a contiguous array of doubles. To check if items are a double, the format attribute is checked to see if the string is "d". This is the same code that the `struct` module uses when encoding binary values. As a general rule, format could be any format string that's compatible with the `struct` module and might include multiple items in the case of arrays containing C structures.

Once we have verified the underlying buffer information, we simply pass it to the C function, which treats it as a normal C array. For all practical purposes, it is not concerned with what kind of array it is or what library created it. This is how the function is able to work with arrays created by the `array` module or by `numpy`.

Before returning a final result, the underlying buffer view must be released using `PyBuffer_Release()`. This step is required to properly manage reference counts of objects.

Again, this recipe only shows a tiny fragment of code that receives an array. If working with arrays, you might run into issues with multidimensional data, strided data, different data types, and more that will require study. Make sure you consult the [official documentation](#) to get more details.

If you need to write many extensions involving array handling, you may find it easier to implement the code in Cython. See [Recipe 15.11](#).

## 15.4. Managing Opaque Pointers in C Extension Modules

### Problem

You have an extension module that needs to handle a pointer to a C data structure, but you don't want to expose any internal details of the structure to Python.

### Solution

Opaque data structures are easily handled by wrapping them inside capsule objects. Consider this fragment of C code from our sample code:

```
typedef struct Point {
    double x,y;
} Point;

extern double distance(Point *p1, Point *p2);
```

Here is an example of extension code that wraps the `Point` structure and `distance()` function using capsules:

```
/* Destructor function for points */
static void del_Point(PyObject *obj) {
    free(PyCapsule_GetPointer(obj, "Point"));
}

/* Utility functions */
static Point *PyPoint_AsPoint(PyObject *obj) {
    return (Point *) PyCapsule_GetPointer(obj, "Point");
}

static PyObject *PyPoint_FromPoint(Point *p, int must_free) {
    return PyCapsule_New(p, "Point", must_free ? del_Point : NULL);
}

/* Create a new Point object */
static PyObject *py_Point(PyObject *self, PyObject *args) {
```

```

Point *p;
double x,y;
if (!PyArg_ParseTuple(args,"dd",&x,&y)) {
    return NULL;
}
p = (Point *) malloc(sizeof(Point));
p->x = x;
p->y = y;
return PyPoint_FromPoint(p, 1);
}

static PyObject *py_distance(PyObject *self, PyObject *args) {
    Point *p1, *p2;
    PyObject *py_p1, *py_p2;
    double result;

    if (!PyArg_ParseTuple(args,"OO",&py_p1, &py_p2)) {
        return NULL;
    }
    if (!(p1 = PyPoint_AsPoint(py_p1))) {
        return NULL;
    }
    if (!(p2 = PyPoint_AsPoint(py_p2))) {
        return NULL;
    }
    result = distance(p1,p2);
    return Py_BuildValue("d", result);
}

```

Using these functions from Python looks like this:

```

>>> import sample
>>> p1 = sample.Point(2,3)
>>> p2 = sample.Point(4,5)
>>> p1
<capsule object "Point" at 0x1004ea330>
>>> p2
<capsule object "Point" at 0x1005d1db0>
>>> sample.distance(p1,p2)
2.8284271247461903
>>>

```

## Discussion

Capsules are similar to a typed C pointer. Internally, they hold a generic pointer along with an identifying name and can be easily created using the `PyCapsule_New()` function. In addition, an optional destructor function can be attached to a capsule to release the underlying memory when the capsule object is garbage collected.

To extract the pointer contained inside a capsule, use the `PyCapsule_GetPointer()` function and specify the name. If the supplied name doesn't match that of the capsule or some other error occurs, an exception is raised and `NULL` is returned.

In this recipe, a pair of utility functions—`PyPoint_FromPoint()` and `PyPoint_AsPoint()`—have been written to deal with the mechanics of creating and unwinding `Point` instances from capsule objects. In any extension functions, we'll use these functions instead of working with capsules directly. This design choice makes it easier to deal with possible changes to the wrapping of `Point` objects in the future. For example, if you decided to use something other than a capsule later, you would only have to change these two functions.

One tricky part about capsules concerns garbage collection and memory management. The `PyPoint_FromPoint()` function accepts a `must_free` argument that indicates whether the underlying `Point *` structure is to be collected when the capsule is destroyed. When working with certain kinds of C code, ownership issues can be difficult to handle (e.g., perhaps a `Point` structure is embedded within a larger data structure that is managed separately). Rather than making a unilateral decision to garbage collect, this extra argument gives control back to the programmer. It should be noted that the destructor associated with an existing capsule can also be changed using the `PyCapsule_SetDestructor()` function.

Capsules are a sensible solution to interfacing with certain kinds of C code involving structures. For instance, sometimes you just don't care about exposing the internals of a structure or turning it into a full-fledged extension type. With a capsule, you can put a lightweight wrapper around it and easily pass it around to other extension functions.

## 15.5. Defining and Exporting C APIs from Extension Modules

### Problem

You have a C extension module that internally defines a variety of useful functions that you would like to export as a public C API for use elsewhere. You would like to use these functions inside other extension modules, but don't know how to link them together, and doing it with the C compiler/linker seems excessively complicated (or impossible).

### Solution

This recipe focuses on the code written to handle `Point` objects, which were presented in [Recipe 15.4](#). If you recall, that C code included some utility functions like this:

```
/* Destructor function for points */  
static void del_Point(PyObject *obj) {
```



```

    free(PyCapsule_GetPointer(obj, "Point"));
}

/* Utility functions */
static Point *PyPoint_AsPoint(PyObject *obj) {
    return (Point *) PyCapsule_GetPointer(obj, "Point");
}

static PyObject *PyPoint_FromPoint(Point *p, int must_free) {
    return PyCapsule_New(p, "Point", must_free ? del_Point : NULL);
}

```

The problem now addressed is how to export the `PyPoint_AsPoint()` and `PyPoint_FromPoint()` functions as an API that other extension modules could use and link to (e.g., if you have other extensions that also want to use the wrapped `Point` objects).

To solve this problem, start by introducing a new header file for the “sample” extension called *pysample.h*. Put the following code in it:

```

/* pysample.h */
#include "Python.h"
#include "sample.h"
#ifdef __cplusplus
extern "C" {
#endif

/* Public API Table */
typedef struct {
    Point *(*aspoint)(PyObject *);
    PyObject *(*frompoint)(Point *, int);
} _PointAPIMethods;

#ifndef PYSAMPLE_MODULE
/* Method table in external module */
static _PointAPIMethods *_point_api = 0;

/* Import the API table from sample */
static int import_sample(void) {
    _point_api = (_PointAPIMethods *) PyCapsule_Import("sample._point_api", 0);
    return (_point_api != NULL) ? 1 : 0;
}

/* Macros to implement the programming interface */
#define PyPoint_AsPoint(obj) (_point_api->aspoint)(obj)
#define PyPoint_FromPoint(obj) (_point_api->frompoint)(obj)
#endif

#ifdef __cplusplus
}
#endif

```

The most important feature here is the `_PointAPIMethods` table of function pointers. It will be initialized in the exporting module and found by importing modules.

Change the original extension module to populate the table and export it as follows:

```
/* pysample.c */

#include "Python.h"
#define PYSAMPLE_MODULE
#include "pysample.h"

...
/* Destructor function for points */
static void del_Point(PyObject *obj) {
    printf("Deleting point\n");
    free(PyCapsule_GetPointer(obj, "Point"));
}

/* Utility functions */
static Point *PyPoint_AsPoint(PyObject *obj) {
    return (Point *) PyCapsule_GetPointer(obj, "Point");
}

static PyObject *PyPoint_FromPoint(Point *p, int free) {
    return PyCapsule_New(p, "Point", free ? del_Point : NULL);
}

static _PointAPIMethods _point_api = {
    PyPoint_AsPoint,
    PyPoint_FromPoint
};

...

/* Module initialization function */
PyMODINIT_FUNC
PyInit_sample(void) {
    PyObject *m;
    PyObject *py_point_api;

    m = PyModule_Create(&samplemodule);
    if (m == NULL)
        return NULL;

    /* Add the Point C API functions */
    py_point_api = PyCapsule_New((void *) &_point_api, "sample._point_api", NULL);
    if (py_point_api) {
        PyModule_AddObject(m, "_point_api", py_point_api);
    }
    return m;
}
```

Finally, here is an example of a new extension module that loads and uses these API functions:

```
/* ptexample.c */

/* Include the header associated with the other module */
#include "pysample.h"

/* An extension function that uses the exported API */
static PyObject *print_point(PyObject *self, PyObject *args) {
    PyObject *obj;
    Point *p;
    if (!PyArg_ParseTuple(args, "O", &obj)) {
        return NULL;
    }

    /* Note: This is defined in a different module */
    p = PyPoint_AsPoint(obj);
    if (!p) {
        return NULL;
    }
    printf("%f %f\n", p->x, p->y);
    return Py_BuildValue("");
}

static PyMethodDef PtExampleMethods[] = {
    {"print_point", print_point, METH_VARARGS, "output a point"},
    { NULL, NULL, 0, NULL}
};

static struct PyModuleDef ptexamplemodule = {
    PyModuleDef_HEAD_INIT,
    "ptexample",          /* name of module */
    "A module that imports an API", /* Doc string (may be NULL) */
    -1,                   /* Size of per-interpreter state or -1 */
    PtExampleMethods      /* Method table */
};

/* Module initialization function */
PyMODINIT_FUNC
PyInit_ptexample(void) {
    PyObject *m;

    m = PyModule_Create(&ptexamplemodule);
    if (m == NULL)
        return NULL;

    /* Import sample, loading its API functions */
    if (!import_sample()) {
        return NULL;
    }
}
```

```

    return m;
}

```

When compiling this new module, you don't even need to bother to link against any of the libraries or code from the other module. For example, you can just make a simple *setup.py* file like this:

```

# setup.py
from distutils.core import setup, Extension

setup(name='ptexample',
      ext_modules=[
          Extension('ptexample',
                  ['ptexample.c'],
                  include_dirs = [], # May need pysample.h directory
                  )
      ]
)

```

If it all works, you'll find that your new extension function works perfectly with the C API functions defined in the other module:

```

>>> import sample
>>> p1 = sample.Point(2,3)
>>> p1
<capsule object "Point *" at 0x1004ea330>
>>> import ptexample
>>> ptexample.print_point(p1)
2.000000 3.000000
>>>

```

## Discussion

This recipe relies on the fact that capsule objects can hold a pointer to anything you wish. In this case, the defining module populates a structure of function pointers, creates a capsule that points to it, and saves the capsule in a module-level attribute (e.g., `sample._point_api`).

Other modules can be programmed to pick up this attribute when imported and extract the underlying pointer. In fact, Python provides the `PyCapsule_Import()` utility function, which takes care of all the steps for you. You simply give it the name of the attribute (e.g., `sample._point_api`), and it will find the capsule and extract the pointer all in one step.

There are some C programming tricks involved in making exported functions look normal in other modules. In the *pysample.h* file, a pointer `_point_api` is used to point to the method table that was initialized in the exporting module. A related function `import_sample()` is used to perform the required capsule import and initialize this pointer. This function must be called before any functions are used. Normally, it would

be called in during module initialization. Finally, a set of C preprocessor macros have been defined to transparently dispatch the API functions through the method table. The user just uses the original function names, but doesn't know about the extra indirection through these macros.

Finally, there is another important reason why you might use this technique to link modules together—it's actually easier and it keeps modules more cleanly decoupled. If you didn't want to use this recipe as shown, you might be able to cross-link modules using advanced features of shared libraries and the dynamic loader. For example, putting common API functions into a shared library and making sure that all extension modules link against that shared library. Yes, this works, but it can be tremendously messy in large systems. Essentially, this recipe cuts out all of that magic and allows modules to link to one another through Python's normal import mechanism and just a tiny number of capsule calls. For compilation of modules, you only need to worry about header files, not the hairy details of shared libraries.

Further information about providing C APIs for extension modules can be found [in the Python documentation](#).

## 15.6. Calling Python from C

### Problem

You want to safely execute a Python callable from C and return a result back to C. For example, perhaps you are writing C code that wants to use a Python function as a callback.

### Solution

Calling Python from C is mostly straightforward, but involves a number of tricky parts. The following C code shows an example of how to do it safely:

```
#include <Python.h>

/* Execute func(x,y) in the Python interpreter. The
   arguments and return result of the function must
   be Python floats */

double call_func(PyObject *func, double x, double y) {
    PyObject *args;
    PyObject *kwargs;
    PyObject *result = 0;
    double retval;

    /* Make sure we own the GIL */
    PyGILState_STATE state = PyGILState_Ensure();
```

```

/* Verify that func is a proper callable */
if (!PyCallable_Check(func)) {
    fprintf(stderr, "call_func: expected a callable\n");
    goto fail;
}
/* Build arguments */
args = Py_BuildValue("(dd)", x, y);
kwargs = NULL;

/* Call the function */
result = PyObject_Call(func, args, kwargs);
Py_DECREF(args);
Py_XDECREF(kwargs);

/* Check for Python exceptions (if any) */
if (PyErr_Occurred()) {
    PyErr_Print();
    goto fail;
}

/* Verify the result is a float object */
if (!PyFloat_Check(result)) {
    fprintf(stderr, "call_func: callable didn't return a float\n");
    goto fail;
}

/* Create the return value */
retval = PyFloat_AsDouble(result);
Py_DECREF(result);

/* Restore previous GIL state and return */
PyGILState_Release(state);
return retval;

fail:
Py_XDECREF(result);
PyGILState_Release(state);
abort(); // Change to something more appropriate
}

```

To use this function, you need to have obtained a reference to an existing Python callable to pass in. There are many ways that you can go about doing that, such as having a callable object passed into an extension module or simply writing C code to extract a symbol from an existing module.

Here is a simple example that shows calling a function from an embedded Python interpreter:

```

#include <Python.h>

/* Definition of call_func() same as above */
...

```

```

/* Load a symbol from a module */
PyObject *import_name(const char *modname, const char *symbol) {
    PyObject *u_name, *module;
    u_name = PyUnicode_FromString(modname);
    module = PyImport_Import(u_name);
    Py_DECREF(u_name);
    return PyObject_GetAttrString(module, symbol);
}

/* Simple embedding example */
int main() {
    PyObject *pow_func;
    double x;

    Py_Initialize();
    /* Get a reference to the math.pow function */
    pow_func = import_name("math", "pow");

    /* Call it using our call_func() code */
    for (x = 0.0; x < 10.0; x += 0.1) {
        printf("%0.2f %0.2f\n", x, call_func(pow_func, x, 2.0));
    }
    /* Done */
    Py_DECREF(pow_func);
    Py_Finalize();
    return 0;
}

```

To build this last example, you'll need to compile the C and link against the Python interpreter. Here is a Makefile that shows how you might do it (this is something that might require some amount of fiddling with on your machine):

```

all::
    cc -g embed.c -I/usr/local/include/python3.3m \
        -L/usr/local/lib/python3.3/config-3.3m -lpython3.3m

```

Compiling and running the resulting executable should produce output similar to this:

```

0.00 0.00
0.10 0.01
0.20 0.04
0.30 0.09
0.40 0.16
...

```

Here is a slightly different example that shows an extension function that receives a callable and some arguments and passes them to `call_func()` for the purposes of testing:

```

/* Extension function for testing the C-Python callback */
PyObject *py_call_func(PyObject *self, PyObject *args) {
    PyObject *func;

```

```

double x, y, result;
if (!PyArg_ParseTuple(args, "Odd", &func, &x, &y)) {
    return NULL;
}
result = call_func(func, x, y);
return Py_BuildValue("d", result);
}

```

Using this extension function, you could test it as follows:

```

>>> import sample
>>> def add(x,y):
...     return x+y
...
>>> sample.call_func(add,3,4)
7.0
>>>

```

## Discussion

If you are calling Python from C, the most important thing to keep in mind is that C is generally going to be in charge. That is, C has the responsibility of creating the arguments, calling the Python function, checking for exceptions, checking types, extracting return values, and more.

As a first step, it is critical that you have a Python object representing the callable that you're going to invoke. This could be a function, class, method, built-in method, or anything that implements the `__call__()` operation. To verify that it's callable, use `PyCallable_Check()` as shown in this code fragment:

```

double call_func(PyObject *func, double x, double y) {
    ...
    /* Verify that func is a proper callable */
    if (!PyCallable_Check(func)) {
        fprintf(stderr, "call_func: expected a callable\n");
        goto fail;
    }
    ...
}

```

As an aside, handling errors in the C code is something that you will need to carefully study. As a general rule, you can't just raise a Python exception. Instead, errors will have to be handled in some other manner that makes sense to your C code. In the solution, we're using `goto` to transfer control to an error handling block that calls `abort()`. This causes the whole program to die, but in real code you would probably want to do something more graceful (e.g., return a status code). Keep in mind that C is in charge here, so there isn't anything comparable to just raising an exception. Error handling is something you'll have to engineer into the program somehow.

Calling a function is relatively straightforward—simply use `PyObject_Call()`, supplying it with the callable object, a tuple of arguments, and an optional dictionary of



keyword arguments. To build the argument tuple or dictionary, you can use `Py_BuildValue()`, as shown.

```
double call_func(PyObject *func, double x, double y) {
    PyObject *args;
    PyObject *kwargs;

    ...
    /* Build arguments */
    args = Py_BuildValue("(dd)", x, y);
    kwargs = NULL;

    /* Call the function */
    result = PyObject_Call(func, args, kwargs);
    Py_DECREF(args);
    Py_XDECREF(kwargs);
    ...
}
```

If there are no keyword arguments, you can pass `NULL`, as shown. After making the function call, you need to make sure that you clean up the arguments using `Py_DECREF()` or `Py_XDECREF()`. The latter function safely allows the `NULL` pointer to be passed (which is ignored), which is why we're using it for cleaning up the optional keyword arguments.

After calling the Python function, you must check for the presence of exceptions. The `PyErr_Occurred()` function can be used to do this. Knowing what to do in response to an exception is tricky. Since you're working from C, you really don't have the exception machinery that Python has. Thus, you would have to set an error status code, log the error, or do some kind of sensible processing. In the solution, `abort()` is called for lack of a simpler alternative (besides, hardened C programmers will appreciate the abrupt crash):

```
...
/* Check for Python exceptions (if any) */
if (PyErr_Occurred()) {
    PyErr_Print();
    goto fail;
}
...
fail:
    PyGILState_Release(state);
    abort();
}
```

Extracting information from the return value of calling a Python function is typically going to involve some kind of type checking and value extraction. To do this, you may have to use functions in the **Python concrete objects layer**. In the solution, the code checks for and extracts the value of a Python float using `PyFloat_Check()` and `PyFloat_AsDouble()`.

A final tricky part of calling into Python from C concerns the management of Python's global interpreter lock (GIL). Whenever Python is accessed from C, you need to make sure that the GIL is properly acquired and released. Otherwise, you run the risk of having the interpreter corrupt data or crash. The calls to `PyGILState_Ensure()` and `PyGILState_Release()` make sure that it's done correctly:

```
double call_func(PyObject *func, double x, double y) {
    ...
    double retval;

    /* Make sure we own the GIL */
    PyGILState_STATE state = PyGILState_Ensure();
    ...
    /* Code that uses Python C API functions */
    ...
    /* Restore previous GIL state and return */
    PyGILState_Release(state);
    return retval;

fail:
    PyGILState_Release(state);
    abort();
}
```

Upon return, `PyGILState_Ensure()` always guarantees that the calling thread has exclusive access to the Python interpreter. This is true even if the calling C code is running a different thread that is unknown to the interpreter. At this point, the C code is free to use any Python C-API functions that it wants. Upon successful completion, `PyGILState_Release()` is used to restore the interpreter back to its original state.

It is critical to note that every `PyGILState_Ensure()` call must be followed by a matching `PyGILState_Release()` call—even in cases where errors have occurred. In the solution, the use of a `goto` statement might look like a horrible design, but we're actually using it to transfer control to a common exit block that performs this required step. Think of the code after the `fail:` label as serving the same purpose as code in a Python `finally:` block.

If you write your C code using all of these conventions including management of the GIL, checking for exceptions, and thorough error checking, you'll find that you can reliably call into the Python interpreter from C—even in very complicated programs that utilize advanced programming techniques such as multithreading.

## 15.7. Releasing the GIL in C Extensions

### Problem

You have C extension code in that you want to execute concurrently with other threads in the Python interpreter. To do this, you need to release and reacquire the global interpreter lock (GIL).

### Solution

In C extension code, the GIL can be released and reacquired by inserting the following macros in the code:

```
#include "Python.h"
...

PyObject *pyfunc(PyObject *self, PyObject *args) {
    ...
    Py_BEGIN_ALLOW_THREADS
    // Threaded C code. Must not use Python API functions
    ...
    Py_END_ALLOW_THREADS
    ...
    return result;
}
```

### Discussion

The GIL can only safely be released if you can guarantee that no Python C API functions will be executed in the C code. Typical examples where the GIL might be released are in computationally intensive code that performs calculations on C arrays (e.g., in extensions such as `numpy`) or in code where blocking I/O operations are going to be performed (e.g., reading or writing on a file descriptor).

While the GIL is released, other Python threads are allowed to execute in the interpreter. The `Py_END_ALLOW_THREADS` macro blocks execution until the calling threads reacquires the GIL in the interpreter.

## 15.8. Mixing Threads from C and Python

### Problem

You have a program that involves a mix of C, Python, and threads, but some of the threads are created from C outside the control of the Python interpreter. Moreover, certain threads utilize functions in the Python C API.

## Solution

If you're going to mix C, Python, and threads together, you need to make sure you properly initialize and manage Python's global interpreter lock (GIL). To do this, include the following code somewhere in your C code and make sure it's called prior to creation of any threads:

```
#include <Python.h>

...
if (!PyEval_ThreadsInitialized()) {
    PyEval_InitThreads();
}
...
```

For any C code that involves Python objects or the Python C API, make sure you properly acquire and release the GIL first. This is done using `PyGILState_Ensure()` and `PyGILState_Release()`, as shown in the following:

```
...
/* Make sure we own the GIL */
PyGILState_STATE state = PyGILState_Ensure();

/* Use functions in the interpreter */
...
/* Restore previous GIL state and return */
PyGILState_Release(state);
...
```

Every call to `PyGILState_Ensure()` must have a matching call to `PyGILState_Release()`.

## Discussion

In advanced applications involving C and Python, it is not uncommon to have many things going on at once—possibly involving a mix of a C code, Python code, C threads, and Python threads. As long as you diligently make sure the interpreter is properly initialized and that C code involving the interpreter has the proper GIL management calls, it all should work.

Be aware that the `PyGILState_Ensure()` call does not immediately preempt or interrupt the interpreter. If other code is currently executing, this function will block until that code decides to release the GIL. Internally, the interpreter performs periodic thread switching, so even if another thread is executing, the caller will eventually get to run (although it may have to wait for a while first).

## 15.9. Wrapping C Code with Swig

### Problem

You have existing C code that you would like to access as a C extension module. You would like to do this using the **Swig wrapper generator**.

### Solution

Swig operates by parsing C header files and automatically creating extension code. To use it, you first need to have a C header file. For example, this header file for our sample code:

```
/* sample.h */

#include <math.h>
extern int gcd(int, int);
extern int in_mandel(double x0, double y0, int n);
extern int divide(int a, int b, int *remainder);
extern double avg(double *a, int n);

typedef struct Point {
    double x,y;
} Point;

extern double distance(Point *p1, Point *p2);
```

Once you have the header files, the next step is to write a Swig “interface” file. By convention, these files have a *.i* suffix and might look similar to the following:

```
// sample.i - Swig interface
%module sample
%{
#include "sample.h"
%}

/* Customizations */
%extend Point {
    /* Constructor for Point objects */
    Point(double x, double y) {
        Point *p = (Point *) malloc(sizeof(Point));
        p->x = x;
        p->y = y;
        return p;
    };
};

/* Map int *remainder as an output argument */
#include typemaps.i
%apply int *OUTPUT { int * remainder };
```

```

/* Map the argument pattern (double *a, int n) to arrays */
%typemap(in) (double *a, int n)(Py_buffer view) {
    view.obj = NULL;
    if (PyObject_GetBuffer($input, &view, PyBUF_ANY_CONTIGUOUS | PyBUF_FORMAT) == -1) {
        SWIG_fail;
    }
    if (strcmp(view.format, "d") != 0) {
        PyErr_SetString(PyExc_TypeError, "Expected an array of doubles");
        SWIG_fail;
    }
    $1 = (double *) view.buf;
    $2 = view.len / sizeof(double);
}

%typemap(freearg) (double *a, int n) {
    if (view$argnum.obj) {
        PyBuffer_Release(&view$argnum);
    }
}

/* C declarations to be included in the extension module */

extern int gcd(int, int);
extern int in_mandel(double x0, double y0, int n);
extern int divide(int a, int b, int *remainder);
extern double avg(double *a, int n);

typedef struct Point {
    double x,y;
} Point;

extern double distance(Point *p1, Point *p2);

```

Once you have written the interface file, Swig is invoked as a command-line tool:

```

bash % swig -python -py3 sample.i
bash %

```

The output of swig is two files, *sample\_wrap.c* and *sample.py*. The latter file is what users import. The *sample\_wrap.c* file is C code that needs to be compiled into a supporting module called `_sample`. This is done using the same techniques as for normal extension modules. For example, you create a *setup.py* file like this:

```

# setup.py
from distutils.core import setup, Extension

setup(name='sample',
      py_modules=['sample.py'],
      ext_modules=[
          Extension('_sample',
                  ['sample_wrap.c'],
                  include_dirs = [],
                  define_macros = [],

```

```

        undef_macros = [],
        library_dirs = [],
        libraries = ['sample']
    )
]
)

```

To compile and test, run python3 on the *setup.py* file like this:

```

bash % python3 setup.py build_ext --inplace
running build_ext
building '_sample' extension
gcc -fno-strict-aliasing -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-prototypes
-I/usr/local/include/python3.3m -c sample_wrap.c
-o build/temp.macosx-10.6-x86_64-3.3/sample_wrap.o
sample_wrap.c: In function 'SWIG_InitializeModule':
sample_wrap.c:3589: warning: statement with no effect
gcc -bundle -undefined dynamic_lookup build/temp.macosx-10.6-x86_64-3.3/sample.o
build/temp.macosx-10.6-x86_64-3.3/sample_wrap.o -o _sample.so -lsample
bash %

```

If all of this works, you'll find that you can use the resulting C extension module in a straightforward way. For example:

```

>>> import sample
>>> sample.gcd(42,8)
2
>>> sample.divide(42,8)
[5, 2]
>>> p1 = sample.Point(2,3)
>>> p2 = sample.Point(4,5)
>>> sample.distance(p1,p2)
2.8284271247461903
>>> p1.x
2.0
>>> p1.y
3.0
>>> import array
>>> a = array.array('d',[1,2,3])
>>> sample.avg(a)
2.0
>>>

```

## Discussion

Swig is one of the oldest tools for building extension modules, dating back to Python 1.4. However, recent versions currently support Python 3. The primary users of Swig tend to have large existing bases of C that they are trying to access using Python as a high-level control language. For instance, a user might have C code containing thousands of functions and various data structures that they would like to access from Python. Swig can automate much of the wrapper generation process.

All Swig interfaces tend to start with a short preamble like this:

```
%module sample
%{
#include "sample.h"
%}
```

This merely declares the name of the extension module and specifies C header files that must be included to make everything compile (the code enclosed in `%{` and `%}` is pasted directly into the output code so this is where you put all included files and other definitions needed for compilation).

The bottom part of a Swig interface is a listing of C declarations that you want to be included in the extension. This is often just copied from the header files. In our example, we just pasted in the header file directly like this:

```
%module sample
%{
#include "sample.h"
%}
...
extern int gcd(int, int);
extern int in_mandel(double x0, double y0, int n);
extern int divide(int a, int b, int *remainder);
extern double avg(double *a, int n);

typedef struct Point {
    double x,y;
} Point;

extern double distance(Point *p1, Point *p2);
```

It is important to stress that these declarations are telling Swig what you want to include in the Python module. It is quite common to edit the list of declarations or to make modifications as appropriate. For example, if you didn't want certain declarations to be included, you would remove them from the declaration list.

The most complicated part of using Swig is the various customizations that it can apply to the C code. This is a huge topic that can't be covered in great detail here, but a number of such customizations are shown in this recipe.

The first customization involving the `%extend` directive allows methods to be attached to existing structure and class definitions. In the example, this is used to add a constructor method to the `Point` structure. This customization makes it possible to use the structure like this:

```
>>> p1 = sample.Point(2,3)
>>>
```

If omitted, then `Point` objects would have to be created in a much more clumsy manner like this:



```

>>> # Usage if %extend Point is omitted
>>> p1 = sample.Point()
>>> p1.x = 2.0
>>> p1.y = 3

```

The second customization involving the inclusion of the `typemaps.i` library and the `%apply` directive is instructing Swig that the argument signature `int *remainder` is to be treated as an output value. This is actually a pattern matching rule. In all declarations that follow, any time `int *remainder` is encountered, it is handled as output. This customization is what makes the `divide()` function return two values:

```

>>> sample.divide(42,8)
[5, 2]
>>>

```

The last customization involving the `%typemap` directive is probably the most advanced feature shown here. A `typemap` is a rule that gets applied to specific argument patterns in the input. In this recipe, a `typemap` has been written to match the argument pattern `(double *a, int n)`. Inside the `typemap` is a fragment of C code that tells Swig how to convert a Python object into the associated C arguments. The code in this recipe has been written using Python's buffer protocol in an attempt to match any input argument that looks like an array of doubles (e.g., NumPy arrays, arrays created by the `array` module, etc.). See [Recipe 15.3](#).

Within the `typemap` code, substitutions such as `$1` and `$2` refer to variables that hold the converted values of the C arguments in the `typemap` pattern (e.g., `$1` maps to `double *a` and `$2` maps to `int n`). `$input` refers to a `PyObject *` argument that was supplied as an input argument. `$argnum` is the argument number.

Writing and understanding `typemaps` is often the bane of programmers using Swig. Not only is the code rather cryptic, but you need to understand the intricate details of both the Python C API and the way in which Swig interacts with it. The Swig documentation has many more examples and detailed information.

Nevertheless, if you have a lot of a C code to expose as an extension module, Swig can be a very powerful tool for doing it. The key thing to keep in mind is that Swig is basically a compiler that processes C declarations, but with a powerful pattern matching and customization component that lets you change the way in which specific declarations and types get processed. More information can be found at [Swig's website](#), including [Python-specific documentation](#).

## 15.10. Wrapping Existing C Code with Cython

### Problem

You want to use **Cython** to make a Python extension module that wraps around an existing C library.

### Solution

Making an extension module with Cython looks somewhat similar to writing a hand-written extension, in that you will be creating a collection of wrapper functions. However, unlike previous recipes, you won't be doing this in C—the code will look a lot more like Python.

As preliminaries, assume that the sample code shown in the introduction to this chapter has been compiled into a C library called `libsample`. Start by creating a file named *csample.pxd* that looks like this:

```
# csample.pxd
#
# Declarations of "external" C functions and structures

cdef extern from "sample.h":
    int gcd(int, int)
    bint in_mandel(double, double, int)
    int divide(int, int, int *)
    double avg(double *, int) nogil

    ctypedef struct Point:
        double x
        double y

    double distance(Point *, Point *)
```

This file serves the same purpose in Cython as a C header file. The initial declaration `cdef extern from "sample.h"` declares the required C header file. Declarations that follow are taken from that header. The name of this file is *csample.pxd*, not *sample.pxd*—this is important.

Next, create a file named *sample.pyx*. This file will define wrappers that bridge the Python interpreter to the underlying C code declared in the *csample.pxd* file:

```
# sample.pyx

# Import the low-level C declarations
cimport csample

# Import some functionality from Python and the C stdlib
from cpython.pycapsule cimport *
```

```

from libc.stdlib cimport malloc, free

# Wrappers
def gcd(unsigned int x, unsigned int y):
    return csample.gcd(x, y)

def in_mandel(x, y, unsigned int n):
    return csample.in_mandel(x, y, n)

def divide(x, y):
    cdef int rem
    quot = csample.divide(x, y, &rem)
    return quot, rem

def avg(double[:] a):
    cdef:
        int sz
        double result

    sz = a.size
    with nogil:
        result = csample.avg(<double *> &a[0], sz)
    return result

# Destructor for cleaning up Point objects
cdef del_Point(object obj):
    pt = <csample.Point *> PyCapsule_GetPointer(obj, "Point")
    free(<void *> pt)

# Create a Point object and return as a capsule
def Point(double x, double y):
    cdef csample.Point *p
    p = <csample.Point *> malloc(sizeof(csample.Point))
    if p == NULL:
        raise MemoryError("No memory to make a Point")
    p.x = x
    p.y = y
    return PyCapsule_New(<void *>p, "Point", <PyCapsule_Destructor>del_Point)

def distance(p1, p2):
    pt1 = <csample.Point *> PyCapsule_GetPointer(p1, "Point")
    pt2 = <csample.Point *> PyCapsule_GetPointer(p2, "Point")
    return csample.distance(pt1, pt2)

```

Various details of this file will be covered further in the discussion section. Finally, to build the extension module, create a *setup.py* file that looks like this:

```

from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

ext_modules = [
    Extension('sample',

```

```

        ['sample.pyx'],
        libraries=['sample'],
        library_dirs=['.'])]
setup(
    name = 'Sample extension module',
    cmdclass = {'build_ext': build_ext},
    ext_modules = ext_modules
)

```

To build the resulting module for experimentation, type this:

```

bash % python3 setup.py build_ext --inplace
running build_ext
cythoning sample.pyx to sample.c
building 'sample' extension
gcc -fno-strict-aliasing -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-prototypes
-I/usr/local/include/python3.3m -c sample.c
-o build/temp.macosx-10.6-x86_64-3.3/sample.o
gcc -bundle -undefined dynamic_lookup build/temp.macosx-10.6-x86_64-3.3/sample.o
-L. -lsample -o sample.so
bash %

```

If it works, you should have an extension module `sample.so` that can be used as shown in the following example:

```

>>> import sample
>>> sample.gcd(42,10)
2
>>> sample.in_mandel(1,1,400)
False
>>> sample.in_mandel(0,0,400)
True
>>> sample.divide(42,10)
(4, 2)
>>> import array
>>> a = array.array('d',[1,2,3])
>>> sample.avg(a)
2.0
>>> p1 = sample.Point(2,3)
>>> p2 = sample.Point(4,5)
>>> p1
<capsule object "Point" at 0x1005d1e70>
>>> p2
<capsule object "Point" at 0x1005d1ea0>
>>> sample.distance(p1,p2)
2.8284271247461903
>>>

```

## Discussion

This recipe incorporates a number of advanced features discussed in prior recipes, including manipulation of arrays, wrapping opaque pointers, and releasing the GIL. Each of these parts will be discussed in turn, but it may help to review earlier recipes first.

At a high level, using Cython is modeled after C. The *.pxd* files merely contain C definitions (similar to *.h* files) and the *.pyx* files contain implementation (similar to a *.c* file). The `cimport` statement is used by Cython to import definitions from a *.pxd* file. This is different than using a normal Python `import` statement, which would load a regular Python module.

Although *.pxd* files contain definitions, they are not used for the purpose of automatically creating extension code. Thus, you still have to write simple wrapper functions. For example, even though the *csample.pxd* file declares `int gcd(int, int)` as a function, you still have to write a small wrapper for it in *sample.pyx*. For instance:

```
cimport csample

def gcd(unsigned int x, unsigned int y):
    return csample.gcd(x,y)
```

For simple functions, you don't have to do too much. Cython will generate wrapper code that properly converts the arguments and return value. The C data types attached to the arguments are optional. However, if you include them, you get additional error checking for free. For example, if someone calls this function with negative values, an exception is generated:

```
>>> sample.gcd(-10,2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "sample.pyx", line 7, in sample.gcd (sample.c:1284)
    def gcd(unsigned int x,unsigned int y):
OverflowError: can't convert negative value to unsigned int
>>>
```

If you want to add additional checking to the wrapper, just use additional wrapper code. For example:

```
def gcd(unsigned int x, unsigned int y):
    if x <= 0:
        raise ValueError("x must be > 0")
    if y <= 0:
        raise ValueError("y must be > 0")
    return csample.gcd(x,y)
```

The declaration of `in_mandel()` in the *csample.pxd* file has an interesting, but subtle definition. In that file, the function is declared as returning a `bint` instead of an `int`. This causes the function to create a proper Boolean value from the result instead of a simple integer. So, a return value of 0 gets mapped to `False` and 1 to `True`.

Within the Cython wrappers, you have the option of declaring C data types in addition to using all of the usual Python objects. The wrapper for `divide()` shows an example of this as well as how to handle a pointer argument.

```
def divide(x,y):
    cdef int rem
    quot = csample.divide(x,y,&rem)
    return quot, rem
```

Here, the `rem` variable is explicitly declared as a C `int` variable. When passed to the underlying `divide()` function, `&rem` makes a pointer to it just as in C.

The code for the `avg()` function illustrates some more advanced features of Cython. First the declaration `def avg(double[: ] a)` declares `avg()` as taking a one-dimensional memoryview of `double` values. The amazing part about this is that the resulting function will accept any compatible array object, including those created by libraries such as `numpy`. For example:

```
>>> import array
>>> a = array.array('d',[1,2,3])
>>> import numpy
>>> b = numpy.array([1., 2., 3.])
>>> import sample
>>> sample.avg(a)
2.0
>>> sample.avg(b)
2.0
>>>
```

In the wrapper, `a.size` and `&a[0]` refer to the number of array items and underlying pointer, respectively. The syntax `<double *> &a[0]` is how you type cast pointers to a different type if necessary. This is needed to make sure the C `avg()` receives a pointer of the correct type. Refer to the next recipe for some more advanced usage of Cython memoryviews.

In addition to working with general arrays, the `avg()` example also shows how to work with the global interpreter lock. The statement `with nogil:` declares a block of code as executing without the GIL. Inside this block, it is illegal to work with any kind of normal Python object—only objects and functions declared as `cdef` can be used. In addition to that, external functions must explicitly declare that they can execute without the GIL. Thus, in the `csample.pxd` file, the `avg()` is declared as `double avg(double *, int) nogil`.

The handling of the `Point` structure presents a special challenge. As shown, this recipe treats `Point` objects as opaque pointers using capsule objects, as described in [Recipe 15.4](#). However, to do this, the underlying Cython code is a bit more complicated. First, the following imports are being used to bring in definitions of functions from the C library and Python C API:

```
from cpython.pycapsule cimport *
from libc.stdlib cimport malloc, free
```

The function `del_Point()` and `Point()` use this functionality to create a capsule object that wraps around a `Point *` pointer. The declaration `cdef del_Point()` declares `del_Point()` as a function that is only accessible from Cython and not Python. Thus, this function will not be visible to the outside—instead, it’s used as a callback function to clean up memory allocated by the capsule. Calls to functions such as `PyCapsule_New()`, `PyCapsule_GetPointer()` are directly from the Python C API and are used in the same way.

The `distance()` function has been written to extract pointers from the capsule objects created by `Point()`. One notable thing here is that you simply don’t have to worry about exception handling. If a bad object is passed, `PyCapsule_GetPointer()` raises an exception, but Cython already knows to look for it and propagate it out of the `distance()` function if it occurs.

A downside to the handling of `Point` structures is that they will be completely opaque in this implementation. You won’t be able to peek inside or access any of their attributes. There is an alternative approach to wrapping, which is to define an extension type, as shown in this code:

```
# sample.pyx

cimport csample
from libc.stdlib cimport malloc, free
...

cdef class Point:
    cdef csample.Point *_c_point
    def __cinit__(self, double x, double y):
        self._c_point = <csample.Point *> malloc(sizeof(csample.Point))
        self._c_point.x = x
        self._c_point.y = y

    def __dealloc__(self):
        free(self._c_point)

    property x:
        def __get__(self):
            return self._c_point.x
        def __set__(self, value):
            self._c_point.x = value

    property y:
        def __get__(self):
            return self._c_point.y
        def __set__(self, value):
            self._c_point.y = value
```

```
def distance(Point p1, Point p2):
    return csample.distance(p1._c_point, p2._c_point)
```

Here, the `cdef class Point` is declaring `Point` as an extension type. The class variable `cdef csample.Point *_c_point` is declaring an instance variable that holds a pointer to an underlying `Point` structure in C. The `__cinit__()` and `__dealloc__()` methods create and destroy the underlying C structure using `malloc()` and `free()` calls. The property `x` and property `y` declarations give code that gets and sets the underlying structure attributes. The wrapper for `distance()` has also been suitably modified to accept instances of the `Point` extension type as arguments, but pass the underlying pointer to the C function.

Making this change, you will find that the code for manipulating `Point` objects is more natural:

```
>>> import sample
>>> p1 = sample.Point(2,3)
>>> p2 = sample.Point(4,5)
>>> p1
<sample.Point object at 0x100447288>
>>> p2
<sample.Point object at 0x1004472a0>
>>> p1.x
2.0
>>> p1.y
3.0
>>> sample.distance(p1,p2)
2.8284271247461903
>>>
```

This recipe has illustrated many of Cython's core features that you might be able to extrapolate to more complicated kinds of wrapping. However, you will definitely want to read more of the [official documentation](#) to do more.

The next few recipes also illustrate a few additional Cython features.

## 15.11. Using Cython to Write High-Performance Array Operations

### Problem

You would like to write some high-performance array processing functions to operate on arrays from libraries such as NumPy. You've heard that tools such as Cython can make this easier, but aren't sure how to do it.



## Solution

As an example, consider the following code which shows a Cython function for clipping the values in a simple one-dimensional array of doubles:

```
# sample.pyx (Cython)

cimport cython

@cython.boundscheck(False)
@cython.wraparound(False)
cpdef clip(double[:] a, double min, double max, double[:] out):
    """
    Clip the values in a to be between min and max. Result in out
    """
    if min > max:
        raise ValueError("min must be <= max")
    if a.shape[0] != out.shape[0]:
        raise ValueError("input and output arrays must be the same size")
    for i in range(a.shape[0]):
        if a[i] < min:
            out[i] = min
        elif a[i] > max:
            out[i] = max
        else:
            out[i] = a[i]
```

To compile and build the extension, you'll need a *setup.py* file such as the following (use `python3 setup.py build_ext --inplace` to build it):

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

ext_modules = [
    Extension('sample',
        ['sample.pyx'])
]

setup(
    name = 'Sample app',
    cmdclass = {'build_ext': build_ext},
    ext_modules = ext_modules
)
```

You will find that the resulting function clips arrays, and that it works with many different kinds of array objects. For example:

```
>>> # array module example
>>> import sample
>>> import array
>>> a = array.array('d',[1,-3,4,7,2,0])
>>> a
```

```

array('d', [1.0, -3.0, 4.0, 7.0, 2.0, 0.0])
>>> sample.clip(a,1,4,a)
>>> a
array('d', [1.0, 1.0, 4.0, 4.0, 2.0, 1.0])

>>> # numpy example
>>> import numpy
>>> b = numpy.random.uniform(-10,10,size=1000000)
>>> b
array([-9.55546017,  7.45599334,  0.69248932, ...,  0.69583148,
        -3.86290931,  2.37266888])
>>> c = numpy.zeros_like(b)
>>> c
array([ 0.,  0.,  0., ...,  0.,  0.,  0.])
>>> sample.clip(b,-5,5,c)
>>> c
array([-5.,  5.,  0.69248932, ...,  0.69583148,
        -3.86290931,  2.37266888])
>>> min(c)
-5.0
>>> max(c)
5.0
>>>

```

You will also find that the resulting code is fast. The following session puts our implementation in a head-to-head battle with the `clip()` function already present in `numpy`:

```

>>> timeit('numpy.clip(b,-5,5,c)','from __main__ import b,c,numpy',number=1000)
8.093049556000551
>>> timeit('sample.clip(b,-5,5,c)','from __main__ import b,c,sample',
...       number=1000)
3.760528204000366
>>>

```

As you can see, it's quite a bit faster—an interesting result considering the core of the NumPy version is written in C.

## Discussion

This recipe utilizes Cython typed memoryviews, which greatly simplify code that operates on arrays. The declaration `cpdef clip()` declares `clip()` as both a C-level and Python-level function. In Cython, this is useful, because it means that the function call is more efficiently called by other Cython functions (e.g., if you want to invoke `clip()` from a different Cython function).

The typed parameters `double[:] a` and `double[:] out` declare those parameters as one-dimensional arrays of doubles. As input, they will access any array object that properly implements the memoryview interface, as described in [PEP 3118](#). This includes arrays from NumPy and from the built-in array library.

When writing code that produces a result that is also an array, you should follow the convention shown of having an output parameter as shown. This places the responsibility of creating the output array on the caller and frees the code from having to know too much about the specific details of what kinds of arrays are being manipulated (it just assumes the arrays are already in-place and only needs to perform a few basic sanity checks such as making sure their sizes are compatible). In libraries such as NumPy, it is relatively easy to create output arrays using functions such as `numpy.zeros()` or `numpy.zeros_like()`. Alternatively, to create uninitialized arrays, you can use `numpy.empty()` or `numpy.empty_like()`. This will be slightly faster if you're about to overwrite the array contents with a result.

In the implementation of your function, you simply write straightforward looking array processing code using indexing and array lookups (e.g., `a[i]`, `out[i]`, and so forth). Cython will take steps to make sure these produce efficient code.

The two decorators that precede the definition of `clip()` are a few optional performance optimizations. `@cython.boundscheck(False)` eliminates all array bounds checking and can be used if you know the indexing won't go out of range. `@cython.wraparound(False)` eliminates the handling of negative array indices as wrapping around to the end of the array (like with Python lists). The inclusion of these decorators can make the code run substantially faster (almost 2.5 times faster on this example when tested).

Whenever working with arrays, careful study and experimentation with the underlying algorithm can also yield large speedups. For example, consider this variant of the `clip()` function that uses conditional expressions:

```
@cython.boundscheck(False)
@cython.wraparound(False)
cpdef clip(double[:] a, double min, double max, double[:] out):
    if min > max:
        raise ValueError("min must be <= max")
    if a.shape[0] != out.shape[0]:
        raise ValueError("input and output arrays must be the same size")
    for i in range(a.shape[0]):
        out[i] = (a[i] if a[i] < max else max) if a[i] > min else min
```

When tested, this version of the code runs over 50% faster (2.44s versus 3.76s on the `timeit()` test shown earlier).

At this point, you might be wondering how this code would stack up against a hand-written C version. For example, perhaps you write the following C function and craft a handwritten extension to using techniques shown in earlier recipes:

```
void clip(double *a, int n, double min, double max, double *out) {
    double x;
    for (; n >= 0; n--, a++, out++) {
        x = *a;
```

```

        *out = x > max ? max : (x < min ? min : x);
    }
}

```

The extension code for this isn't shown, but after experimenting, we found that a hand-crafted C extension ran more than 10% slower than the version created by Cython. The bottom line is that the code runs a lot faster than you might think.

There are several extensions that can be made to the solution code. For certain kinds of array operations, it might make sense to release the GIL so that multiple threads can run in parallel. To do that, modify the code to include the `with nogil:` statement:

```

@cython.boundscheck(False)
@cython.wraparound(False)
cdef clip(double[:]) a, double min, double max, double[:] out):
    if min > max:
        raise ValueError("min must be <= max")
    if a.shape[0] != out.shape[0]:
        raise ValueError("input and output arrays must be the same size")
    with nogil:
        for i in range(a.shape[0]):
            out[i] = (a[i] if a[i] < max else max) if a[i] > min else min

```

If you want to write a version of the code that operates on two-dimensional arrays, here is what it might look like:

```

@cython.boundscheck(False)
@cython.wraparound(False)
cdef clip2d(double[:,:] a, double min, double max, double[:,:] out):
    if min > max:
        raise ValueError("min must be <= max")
    for n in range(a.ndim):
        if a.shape[n] != out.shape[n]:
            raise TypeError("a and out have different shapes")
    for i in range(a.shape[0]):
        for j in range(a.shape[1]):
            if a[i,j] < min:
                out[i,j] = min
            elif a[i,j] > max:
                out[i,j] = max
            else:
                out[i,j] = a[i,j]

```

Hopefully it's not lost on the reader that all of the code in this recipe is not tied to any specific array library (e.g., NumPy). That gives the code a great deal of flexibility. However, it's also worth noting that dealing with arrays can be significantly more complicated once multiple dimensions, strides, offsets, and other factors are introduced. Those topics are beyond the scope of this recipe, but more information can be found in [PEP 3118](#). The [Cython documentation on “typed memoryviews”](#) is also essential reading.

## 15.12. Turning a Function Pointer into a Callable

### Problem

You have (somehow) obtained the memory address of a compiled function, but want to turn it into a Python callable that you can use as an extension function.

### Solution

The `ctypes` module can be used to create Python callables that wrap around arbitrary memory addresses. The following example shows how to obtain the raw, low-level address of a C function and how to turn it back into a callable object:

```
>>> import ctypes
>>> lib = ctypes.cdll.LoadLibrary(None)
>>> # Get the address of sin() from the C math library
>>> addr = ctypes.cast(lib.sin, ctypes.c_void_p).value
>>> addr
140735505915760

>>> # Turn the address into a callable function
>>> func_type = ctypes.CFUNCTYPE(ctypes.c_double, ctypes.c_double)
>>> func = func_type(addr)
>>> func
<CFunctionType object at 0x1006816d0>

>>> # Call the resulting function
>>> func(2)
0.9092974268256817
>>> func(0)
0.0
>>>
```

### Discussion

To make a callable, you must first create a `CFUNCTYPE` instance. The first argument to `CFUNCTYPE()` is the return type. Subsequent arguments are the types of the arguments. Once you have defined the function type, you wrap it around an integer memory address to create a callable object. The resulting object is used like any normal function accessed through `ctypes`.

This recipe might look rather cryptic and low level. However, it is becoming increasingly common for programs and libraries to utilize advanced code generation techniques like just in-time compilation, as found in libraries such as LLVM.

For example, here is a simple example that uses the `llvmpy extension` to make a small assembly function, obtain a function pointer to it, and turn it into a Python callable:

```

>>> from llvm.core import Module, Function, Type, Builder
>>> mod = Module.new('example')
>>> f = Function.new(mod, Type.function(Type.double(), \
    [Type.double(), Type.double()], False), 'foo')
>>> block = f.append_basic_block('entry')
>>> builder = Builder.new(block)
>>> x2 = builder.fmul(f.args[0], f.args[0])
>>> y2 = builder.fmul(f.args[1], f.args[1])
>>> r = builder.fadd(x2, y2)
>>> builder.ret(r)
<llvm.core.Instruction object at 0x10078e990>
>>> from llvm.ee import ExecutionEngine
>>> engine = ExecutionEngine.new(mod)
>>> ptr = engine.get_pointer_to_function(f)
>>> ptr
4325863440
>>> foo = ctypes.CFUNCTYPE(ctypes.c_double, ctypes.c_double, ctypes.c_double)(ptr)

>>> # Call the resulting function
>>> foo(2, 3)
13.0
>>> foo(4, 5)
41.0
>>> foo(1, 2)
5.0
>>>

```

It goes without saying that doing anything wrong at this level will probably cause the Python interpreter to die a horrible death. Keep in mind that you're directly working with machine-level memory addresses and native machine code—not Python functions.

## 15.13. Passing NULL-Terminated Strings to C Libraries

### Problem

You are writing an extension module that needs to pass a NULL-terminated string to a C library. However, you're not entirely sure how to do it with Python's Unicode string implementation.

### Solution

Many C libraries include functions that operate on NULL-terminated strings declared as type `char *`. Consider the following C function that we will use for the purposes of illustration and testing:

```

void print_chars(char *s) {
    while (*s) {
        printf("%2x ", (unsigned char) *s);
    }
}

```

```

        s++;
    }
    printf("\n");
}

```

This function simply prints out the hex representation of individual characters so that the passed strings can be easily debugged. For example:

```
print_chars("Hello"); // Outputs: 48 65 6c 6c 6f
```

For calling such a C function from Python, you have a few choices. First, you could restrict it to only operate on bytes using "y" conversion code to `PyArg_ParseTuple()` like this:

```

static PyObject *py_print_chars(PyObject *self, PyObject *args) {
    char *s;

    if (!PyArg_ParseTuple(args, "y", &s)) {
        return NULL;
    }
    print_chars(s);
    Py_RETURN_NONE;
}

```

The resulting function operates as follows. Carefully observe how bytes with embedded NULL bytes and Unicode strings are rejected:

```

>>> print_chars(b'Hello World')
48 65 6c 6c 6f 20 57 6f 72 6c 64
>>> print_chars(b'Hello\x00World')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be bytes without null bytes, not bytes
>>> print_chars('Hello World')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' does not support the buffer interface
>>>

```

If you want to pass Unicode strings instead, use the "s" format code to `PyArg_ParseTuple()` such as this:

```

static PyObject *py_print_chars(PyObject *self, PyObject *args) {
    char *s;

    if (!PyArg_ParseTuple(args, "s", &s)) {
        return NULL;
    }
    print_chars(s);
    Py_RETURN_NONE;
}

```

When used, this will automatically convert all strings to a NULL-terminated UTF-8 encoding. For example:

```
>>> print_chars('Hello World')
48 65 6c 6c 6f 20 57 6f 72 6c 64
>>> print_chars('Spicy Jalape\u00f1o') # Note: UTF-8 encoding
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f
>>> print_chars('Hello\x00World')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str without null characters, not str
>>> print_chars(b'Hello World')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not bytes
>>>
```

If for some reason, you are working directly with a PyObject \* and can't use PyArg\_ParseTuple(), the following code samples show how you can check and extract a suitable char \* reference, from both a bytes and string object:

```
/* Some Python Object (obtained somehow) */
PyObject *obj;

/* Conversion from bytes */
{
    char *s;
    s = PyBytes_AsString(o);
    if (!s) {
        return NULL; /* TypeError already raised */
    }
    print_chars(s);
}

/* Conversion to UTF-8 bytes from a string */
{
    PyObject *bytes;
    char *s;
    if (!PyUnicode_Check(obj)) {
        PyErr_SetString(PyExc_TypeError, "Expected string");
        return NULL;
    }
    bytes = PyUnicode_AsUTF8String(obj);
    s = PyBytes_AsString(bytes);
    print_chars(s);
    Py_DECREF(bytes);
}
```

Both of the preceding conversions guarantee NULL-terminated data, but they do not check for embedded NULL bytes elsewhere inside the string. Thus, that's something that you would need to check yourself if it's important.



## Discussion

If it all possible, you should try to avoid writing code that relies on NULL-terminated strings since Python has no such requirement. It is almost always better to handle strings using the combination of a pointer and a size if possible. Nevertheless, sometimes you have to work with legacy C code that presents no other option.

Although it is easy to use, there is a hidden memory overhead associated with using the "s" format code to `PyArg_ParseTuple()` that is easy to overlook. When you write code that uses this conversion, a UTF-8 string is created and permanently attached to the original string object. If the original string contains non-ASCII characters, this makes the size of the string increase until it is garbage collected. For example:

```
>>> import sys
>>> s = 'Spicy Jalape\u00f1o'
>>> sys.getsizeof(s)
87
>>> print_chars(s)      # Passing string
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f
>>> sys.getsizeof(s)    # Notice increased size
103
>>>
```

If this growth in memory use is a concern, you should rewrite your C extension code to use the `PyUnicode_AsUTF8String()` function like this:

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
    PyObject *o, *bytes;
    char *s;

    if (!PyArg_ParseTuple(args, "U", &o)) {
        return NULL;
    }
    bytes = PyUnicode_AsUTF8String(o);
    s = PyBytes_AsString(bytes);
    print_chars(s);
    Py_DECREF(bytes);
    Py_RETURN_NONE;
}
```

With this modification, a UTF-8 encoded string is created if needed, but then discarded after use. Here is the modified behavior:

```
>>> import sys
>>> s = 'Spicy Jalape\u00f1o'
>>> sys.getsizeof(s)
87
>>> print_chars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f
>>> sys.getsizeof(s)
87
>>>
```

If you are trying to pass NULL-terminated strings to functions wrapped via `ctypes`, be aware that `ctypes` only allows bytes to be passed and that it does not check for embedded NULL bytes. For example:

```
>>> import ctypes
>>> lib = ctypes.cdll.LoadLibrary("./libsampl.e.so")
>>> print_chars = lib.print_chars
>>> print_chars.argtypes = (ctypes.c_char_p,)
>>> print_chars(b'Hello World')
48 65 6c 6c 6f 20 57 6f 72 6c 64
>>> print_chars(b'Hello\x00World')
48 65 6c 6c 6f
>>> print_chars('Hello World')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 1: <class 'TypeError'>: wrong type
>>>
```

If you want to pass a string instead of bytes, you need to perform a manual UTF-8 encoding first. For example:

```
>>> print_chars('Hello World'.encode('utf-8'))
48 65 6c 6c 6f 20 57 6f 72 6c 64
>>>
```

For other extension tools (e.g., Swig, Cython), careful study is probably in order should you decide to use them to pass strings to C code.

## 15.14. Passing Unicode Strings to C Libraries

### Problem

You are writing an extension module that needs to pass a Python string to a C library function that may or may not know how to properly handle Unicode.

### Solution

There are many issues to be concerned with here, but the main one is that existing C libraries won't understand Python's native representation of Unicode. Therefore, your challenge is to convert the Python string into a form that can be more easily understood by C libraries.

For the purposes of illustration, here are two C functions that operate on string data and output it for the purposes of debugging and experimentation. One uses bytes provided in the form `char *`, `int`, whereas the other uses wide characters in the form `wchar_t *`, `int`:

```
void print_chars(char *s, int len) {
    int n = 0;
```

```

while (n < len) {
    printf("%2x ", (unsigned char) s[n]);
    n++;
}
printf("\n");
}

void print_wchars(wchar_t *s, int len) {
    int n = 0;
    while (n < len) {
        printf("%x ", s[n]);
        n++;
    }
    printf("\n");
}

```

For the byte-oriented function `print_chars()`, you need to convert Python strings into a suitable byte encoding such as UTF-8. Here is a sample extension function that does this:

```

static PyObject *py_print_chars(PyObject *self, PyObject *args) {
    char *s;
    Py_ssize_t len;

    if (!PyArg_ParseTuple(args, "s#", &s, &len)) {
        return NULL;
    }
    print_chars(s, len);
    Py_RETURN_NONE;
}

```

For library functions that work with the machine native `wchar_t` type, you can write extension code such as this:

```

static PyObject *py_print_wchars(PyObject *self, PyObject *args) {
    wchar_t *s;
    Py_ssize_t len;

    if (!PyArg_ParseTuple(args, "u#", &s, &len)) {
        return NULL;
    }
    print_wchars(s, len);
    Py_RETURN_NONE;
}

```

Here is an interactive session that illustrates how these functions work:

```

>>> s = 'Spicy Jalape\u00f1o'
>>> print_chars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f
>>> print_wchars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 f1 6f
>>>

```

Carefully observe how the byte-oriented function `print_chars()` is receiving UTF-8 encoded data, whereas `print_wchars()` is receiving the Unicode code point values.

## Discussion

Before considering this recipe, you should first study the nature of the C library that you're accessing. For many C libraries, it might make more sense to pass bytes instead of a string. To do that, use this conversion code instead:

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
    char *s;
    Py_ssize_t len;

    /* accepts bytes, bytearray, or other byte-like object */
    if (!PyArg_ParseTuple(args, "y#", &s, &len)) {
        return NULL;
    }
    print_chars(s, len);
    Py_RETURN_NONE;
}
```

If you decide that you still want to pass strings, you need to know that Python 3 uses an adaptable string representation that is not entirely straightforward to map directly to C libraries using the standard types `char *` or `wchar_t *`. See [PEP 393](#) for details. Thus, to present string data to C, some kind of conversion is almost always necessary. The `s#` and `u#` format codes to `PyArg_ParseTuple()` safely perform such conversions.

One potential downside is that such conversions cause the size of the original string object to permanently increase. Whenever a conversion is made, a copy of the converted data is kept and attached to the original string object so that it can be reused later. You can observe this effect:

```
>>> import sys
>>> s = 'Spicy Jalape\u00f1o'
>>> sys.getsizeof(s)
87
>>> print_chars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f
>>> sys.getsizeof(s)
103
>>> print_wchars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 f1 6f
>>> sys.getsizeof(s)
163
>>>
```

For small amounts of string data, this might not matter, but if you're doing large amounts of text processing in extensions, you may want to avoid the overhead. Here is an alternative implementation of the first extension function that avoids these memory inefficiencies:

```

static PyObject *py_print_chars(PyObject *self, PyObject *args) {
    PyObject *obj, *bytes;
    char *s;
    Py_ssize_t len;

    if (!PyArg_ParseTuple(args, "U", &obj)) {
        return NULL;
    }
    bytes = PyUnicode_AsUTF8String(obj);
    PyBytes_AsStringAndSize(bytes, &s, &len);
    print_chars(s, len);
    Py_DECREF(bytes);
    Py_RETURN_NONE;
}

```

Avoiding memory overhead for `wchar_t` handling is much more tricky. Internally, Python stores strings using the most efficient representation possible. For example, strings containing nothing but ASCII are stored as arrays of bytes, whereas strings containing characters in the range U+0000 to U+FFFF use a two-byte representation. Since there isn't a single representation of the data, you can't just cast the internal array to `wchar_t *` and hope that it works. Instead, a `wchar_t` array has to be created and text copied into it. The "u#" format code to `PyArg_ParseTuple()` does this for you at the cost of efficiency (it attaches the resulting copy to the string object).

If you want to avoid this long-term memory overhead, your only real choice is to copy the Unicode data into a temporary array, pass it to the C library function, and then deallocate the array. Here is one possible implementation:

```

static PyObject *py_print_wchars(PyObject *self, PyObject *args) {
    PyObject *obj;
    wchar_t *s;
    Py_ssize_t len;

    if (!PyArg_ParseTuple(args, "U", &obj)) {
        return NULL;
    }
    if ((s = PyUnicode_AsWideCharString(obj, &len)) == NULL) {
        return NULL;
    }
    print_wchars(s, len);
    PyMem_Free(s);
    Py_RETURN_NONE;
}

```

In this implementation, `PyUnicode_AsWideCharString()` creates a temporary buffer of `wchar_t` characters and copies data into it. That buffer is passed to C and then released afterward. As of this writing, there seems to be a possible bug related to this behavior, as described at [the Python issues page](#).

If, for some reason you know that the C library takes the data in a different byte encoding than UTF-8, you can force Python to perform an appropriate conversion using extension code such as the following:

```
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
    char *s = 0;
    int len;
    if (!PyArg_ParseTuple(args, "es#", "encoding-name", &s, &len)) {
        return NULL;
    }
    print_chars(s, len);
    PyMem_Free(s);
    Py_RETURN_NONE;
}
```

Last, but not least, if you want to work directly with the characters in a Unicode string, here is an example that illustrates low-level access:

```
static PyObject *py_print_wchars(PyObject *self, PyObject *args) {
    PyObject *obj;
    int n, len;
    int kind;
    void *data;

    if (!PyArg_ParseTuple(args, "U", &obj)) {
        return NULL;
    }
    if (PyUnicode_READY(obj) < 0) {
        return NULL;
    }

    len = PyUnicode_GET_LENGTH(obj);
    kind = PyUnicode_KIND(obj);
    data = PyUnicode_DATA(obj);

    for (n = 0; n < len; n++) {
        Py_UCS4 ch = PyUnicode_READ(kind, data, n);
        printf("%x ", ch);
    }
    printf("\n");
    Py_RETURN_NONE;
}
```

In this code, the `PyUnicode_KIND()` and `PyUnicode_DATA()` macros are related to the variable-width storage of Unicode, as described in [PEP 393](#). The `kind` variable encodes information about the underlying storage (8-bit, 16-bit, or 32-bit) and `data` points the buffer. In reality, you don't need to do anything with these values as long as you pass them to the `PyUnicode_READ()` macro when extracting characters.

A few final words: when passing Unicode strings from Python to C, you should probably try to make it as simple as possible. If given the choice between an encoding such as

UTF-8 or wide characters, choose UTF-8. Support for UTF-8 seems to be much more common, less trouble-prone, and better supported by the interpreter. Finally, make sure your review the [documentation on Unicode handling](#).

## 15.15. Converting C Strings to Python

### Problem

You want to convert strings from C to Python bytes or a string object.

### Solution

For C strings represented as a pair `char *`, `int`, you must decide whether or not you want the string presented as a raw byte string or as a Unicode string. Byte objects can be built using `Py_BuildValue()` as follows:

```
char *s;      /* Pointer to C string data */
int len;      /* Length of data */

/* Make a bytes object */
PyObject *obj = Py_BuildValue("y#", s, len);
```

If you want to create a Unicode string and you know that `s` points to data encoded as UTF-8, you can use the following:

```
PyObject *obj = Py_BuildValue("s#", s, len);
```

If `s` is encoded in some other known encoding, you can make a string using `PyUnicode_Decode()` as follows:

```
PyObject *obj = PyUnicode_Decode(s, len, "encoding", "errors");

/* Examples */
obj = PyUnicode_Decode(s, len, "latin-1", "strict");
obj = PyUnicode_Decode(s, len, "ascii", "ignore");
```

If you happen to have a wide string represented as a `wchar_t *`, `len` pair, there are a few options. First, you could use `Py_BuildValue()` as follows:

```
wchar_t *w;   /* Wide character string */
int len;      /* Length */

PyObject *obj = Py_BuildValue("u#", w, len);
```

Alternatively, you can use `PyUnicode_FromWideChar()`:

```
PyObject *obj = PyUnicode_FromWideChar(w, len);
```

For wide character strings, no interpretation is made of the character data—it is assumed to be raw Unicode code points which are directly converted to Python.

## Discussion

Conversion of strings from C to Python follow the same principles as I/O. Namely, the data from C must be explicitly decoded into a string according to some codec. Common encodings include ASCII, Latin-1, and UTF-8. If you're not entirely sure of the encoding or the data is binary, you're probably best off encoding the string as bytes instead.

When making an object, Python always copies the string data you provide. If necessary, it's up to you to release the C string afterward (if required). Also, for better reliability, you should try to create strings using both a pointer and a size rather than relying on NULL-terminated data.

## 15.16. Working with C Strings of Dubious Encoding

### Problem

You are converting strings back and forth between C and Python, but the C encoding is of a dubious or unknown nature. For example, perhaps the C data is supposed to be UTF-8, but it's not being strictly enforced. You would like to write code that can handle malformed data in a graceful way that doesn't crash Python or destroy the string data in the process.

### Solution

Here is some C data and a function that illustrates the nature of this problem:

```
/* Some dubious string data (malformed UTF-8) */
const char *sdata = "Spicy Jalape\x3\xb1o\xae";
int slen = 16;

/* Output character data */
void print_chars(char *s, int len) {
    int n = 0;
    while (n < len) {
        printf("%2x ", (unsigned char) s[n]);
        n++;
    }
    printf("\n");
}
```

In this code, the string `sdata` contains a mix of UTF-8 and malformed data. Nevertheless, if a user calls `print_chars(sdata, slen)` in C, it works fine.

Now suppose you want to convert the contents of `sdata` into a Python string. Further suppose you want to later pass that string to the `print_chars()` function through an extension. Here's how to do it in a way that exactly preserves the original data even though there are encoding problems:



```

/* Return the C string back to Python */
static PyObject *py_retstr(PyObject *self, PyObject *args) {
    if (!PyArg_ParseTuple(args, "")) {
        return NULL;
    }
    return PyUnicode_Decode(sdata, slen, "utf-8", "surrogateescape");
}

/* Wrapper for the print_chars() function */
static PyObject *py_print_chars(PyObject *self, PyObject *args) {
    PyObject *obj, *bytes;
    char *s = 0;
    Py_ssize_t len;

    if (!PyArg_ParseTuple(args, "U", &obj)) {
        return NULL;
    }

    if ((bytes = PyUnicode_AsEncodedString(obj, "utf-8", "surrogateescape"))
        == NULL) {
        return NULL;
    }
    PyBytes_AsStringAndSize(bytes, &s, &len);
    print_chars(s, len);
    Py_DECREF(bytes);
    Py_RETURN_NONE;
}

```

If you try these functions from Python, here's what happens:

```

>>> s = retstr()
>>> s
'Spicy Jalapeño\udcaé'
>>> print_chars(s)
53 70 69 63 79 20 4a 61 6c 61 70 65 c3 b1 6f ae
>>>

```

Careful observation will reveal that the malformed string got encoded into a Python string without errors, and that when passed back into C, it turned back into a byte string that exactly encoded the same bytes as the original C string.

## Discussion

This recipe addresses a subtle, but potentially annoying problem with string handling in extension modules. Namely, the fact that C strings in extensions might not follow the strict Unicode encoding/decoding rules that Python normally expects. Thus, it's possible that some malformed C data would pass to Python. A good example might be C strings associated with low-level system calls such as filenames. For instance, what happens if a system call returns a broken string back to the interpreter that can't be properly decoded.

Normally, Unicode errors are often handled by specifying some sort of error policy, such as `strict`, `ignore`, `replace`, or something similar. However, a downside of these policies is that they irreparably destroy the original string content. For example, if the malformed data in the example was decoded using one of these policies, you would get results such as this:

```
>>> raw = b'Spicy Jalape\xc3\xb1\xae'
>>> raw.decode('utf-8','ignore')
'Spicy Jalapeño'
>>> raw.decode('utf-8','replace')
'Spicy Jalapeño?'
>>>
```

The `surrogateescape` error handling policy takes all nondecodable bytes and turns them into the low-half of a surrogate pair (`\udcXX` where `XX` is the raw byte value). For example:

```
>>> raw.decode('utf-8','surrogateescape')
'Spicy Jalapeño\udcae'
>>>
```

Isolated low surrogate characters such as `\udcae` never appear in valid Unicode. Thus, this string is technically an illegal representation. In fact, if you ever try to pass it to functions that perform output, you'll get encoding errors:

```
>>> s = raw.decode('utf-8','surrogateescape')
>>> print(s)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'utf-8' codec can't encode character '\udcae'
in position 14: surrogates not allowed
>>>
```

However, the main point of allowing the surrogate escapes is to allow malformed strings to pass from C to Python and back into C without any data loss. When the string is encoded using `surrogateescape` again, the surrogate characters are turned back into their original bytes. For example:

```
>>> s
'Spicy Jalapeño\udcae'
>>> s.encode('utf-8','surrogateescape')
b'Spicy Jalape\xc3\xb1\xae'
>>>
```

As a general rule, it's probably best to avoid surrogate encoding whenever possible—your code will be much more reliable if it uses proper encodings. However, sometimes there are situations where you simply don't have control over the data encoding and you aren't free to ignore or replace the bad data because other functions may need to use it. This recipe shows how to do it.

As a final note, many of Python's system-oriented functions, especially those related to filenames, environment variables, and command-line options, use surrogate encoding. For example, if you use a function such as `os.listdir()` on a directory containing an undecodable filename, it will be returned as a string with surrogate escapes. See [Recipe 5.15](#) for a related recipe.

[PEP 383](#) has more information about the problem addressed by this recipe and surrogateescape error handling.

## 15.17. Passing Filenames to C Extensions

### Problem

You need to pass filenames to C library functions, but need to make sure the filename has been encoded according to the system's expected filename encoding.

### Solution

To write an extension function that receives a filename, use code such as this:

```
static PyObject *py_get_filename(PyObject *self, PyObject *args) {
    PyObject *bytes;
    char *filename;
    Py_ssize_t len;
    if (!PyArg_ParseTuple(args, "O&", PyUnicode_FSConverter, &bytes)) {
        return NULL;
    }
    PyBytes_AsStringAndSize(bytes, &filename, &len);
    /* Use filename */
    ...

    /* Cleanup and return */
    Py_DECREF(bytes);
    Py_RETURN_NONE;
}
```

If you already have a `PyObject *` that you want to convert as a filename, use code such as the following:

```
PyObject *obj;    /* Object with the filename */
PyObject *bytes;
char *filename;
Py_ssize_t len;

bytes = PyUnicode_EncodeFSDefault(obj);
PyBytes_AsStringAndSize(bytes, &filename, &len);
/* Use filename */
...
```

```
/* Cleanup */
Py_DECREF(bytes);
```

If you need to return a filename back to Python, use the following code:

```
/* Turn a filename into a Python object */

char *filename;      /* Already set */
int filename_len;    /* Already set */

PyObject *obj = PyUnicode_DecodeFSDefaultAndSize(filename, filename_len);
```

## Discussion

Dealing with filenames in a portable way is a tricky problem that is best left to Python. If you use this recipe in your extension code, filenames will be handled in a manner that is consistent with filename handling in the rest of Python. This includes encoding/decoding of bytes, dealing with bad characters, surrogate escapes, and other complications.

# 15.18. Passing Open Files to C Extensions

## Problem

You have an open file object in Python, but need to pass it to C extension code that will use the file.

## Solution

To convert a file to an integer file descriptor, use `PyFile_FromFd()`, as shown:

```
PyObject *fobj;      /* File object (already obtained somehow) */
int fd = PyObject_AsFileDescriptor(fobj);
if (fd < 0) {
    return NULL;
}
```

The resulting file descriptor is obtained by calling the `fileno()` method on `fobj`. Thus, any object that exposes a descriptor in this manner should work (e.g., file, socket, etc.).

Once you have the descriptor, it can be passed to various low-level C functions that expect to work with files.

If you need to convert an integer file descriptor back into a Python object, use `PyFile_FromFd()` as follows:

```
int fd;      /* Existing file descriptor (already open) */
PyObject *fobj = PyFile_FromFd(fd, "filename", "r", -1, NULL, NULL, NULL, 1);
```

The arguments to `PyFile_FromFd()` mirror those of the built-in `open()` function. `NULL` values simply indicate that the default settings for the encoding, errors, and `newline` arguments are being used.

## Discussion

If you are passing file objects from Python to C, there are a few tricky issues to be concerned about. First, Python performs its own I/O buffering through the `io` module. Prior to passing any kind of file descriptor to C, you should first flush the I/O buffers on the associated file objects. Otherwise, you could get data appearing out of order on the file stream.

Second, you need to pay careful attention to file ownership and the responsibility of closing the file in particular. If a file descriptor is passed to C, but still used in Python, you need to make sure C doesn't accidentally close the file. Likewise, if a file descriptor is being turned into a Python file object, you need to be clear about who is responsible for closing it. The last argument to `PyFile_FromFd()` is set to 1 to indicate that Python should close the file.

If you need to make a different kind of file object such as a `FILE *` object from the C standard I/O library using a function such as `fdopen()`, you'll need to be especially careful. Doing so would introduce two completely different I/O buffering layers into the I/O stack (one from Python's `io` module and one from C `stdio`). Operations such as `fclose()` in C could also inadvertently close the file for further use in Python. If given a choice, you should probably make extension code work with the low-level integer file descriptors as opposed to using a higher-level abstraction such as that provided by `<stdio.h>`.

## 15.19. Reading File-Like Objects from C

### Problem

You want to write C extension code that consumes data from any Python file-like object (e.g., normal files, `StringIO` objects, etc.).

### Solution

To consume data on a file-like object, you need to repeatedly invoke its `read()` method and take steps to properly decode the resulting data.

Here is a sample C extension function that merely consumes all of the data on a file-like object and dumps it to standard output so you can see it:

```
#define CHUNK_SIZE 8192
```

```

/* Consume a "file-like" object and write bytes to stdout */
static PyObject *py_consume_file(PyObject *self, PyObject *args) {
    PyObject *obj;
    PyObject *read_meth;
    PyObject *result = NULL;
    PyObject *read_args;

    if (!PyArg_ParseTuple(args, "O", &obj)) {
        return NULL;
    }

    /* Get the read method of the passed object */
    if ((read_meth = PyObject_GetAttrString(obj, "read")) == NULL) {
        return NULL;
    }

    /* Build the argument list to read() */
    read_args = Py_BuildValue("(i)", CHUNK_SIZE);
    while (1) {
        PyObject *data;
        PyObject *enc_data;
        char *buf;
        Py_ssize_t len;

        /* Call read() */
        if ((data = PyObject_Call(read_meth, read_args, NULL)) == NULL) {
            goto final;
        }

        /* Check for EOF */
        if (PySequence_Length(data) == 0) {
            Py_DECREF(data);
            break;
        }

        /* Encode Unicode as Bytes for C */
        if ((enc_data=PyUnicode_AsEncodedString(data, "utf-8", "strict"))==NULL) {
            Py_DECREF(data);
            goto final;
        }

        /* Extract underlying buffer data */
        PyBytes_AsStringAndSize(enc_data, &buf, &len);

        /* Write to stdout (replace with something more useful) */
        write(1, buf, len);

        /* Cleanup */
        Py_DECREF(enc_data);
        Py_DECREF(data);
    }
    result = Py_BuildValue("");
}

```

```

final:
    /* Cleanup */
    Py_DECREF(read_meth);
    Py_DECREF(read_args);
    return result;
}

```

To test the code, try making a file-like object such as a `StringIO` instance and pass it in:

```

>>> import io
>>> f = io.StringIO('Hello\nWorld\n')
>>> import sample
>>> sample.consume_file(f)
Hello
World
>>>

```

## Discussion

Unlike a normal system file, a file-like object is not necessarily built around a low-level file descriptor. Thus, you can't use normal C library functions to access it. Instead, you need to use Python's C API to manipulate the file-like object much like you would in Python.

In the solution, the `read()` method is extracted from the passed object. An argument list is built and then repeatedly passed to `PyObject_Call()` to invoke the method. To detect end-of-file (EOF), `PySequence_Length()` is used to see if the returned result has zero length.

For all I/O operations, you'll need to concern yourself with the underlying encoding and distinction between bytes and Unicode. This recipe shows how to read a file in text mode and decode the resulting text into a bytes encoding that can be used by C. If you want to read the file in binary mode, only minor changes will be made. For example:

```

...
/* Call read() */
if ((data = PyObject_Call(read_meth, read_args, NULL)) == NULL) {
    goto final;
}

/* Check for EOF */
if (PySequence_Length(data) == 0) {
    Py_DECREF(data);
    break;
}
if (!PyBytes_Check(data)) {
    Py_DECREF(data);
    PyErr_SetString(PyExc_IOError, "File must be in binary mode");
    goto final;
}

```

```

/* Extract underlying buffer data */
PyBytes_AsStringAndSize(data, &buf, &len);
...

```

The trickiest part of this recipe concerns proper memory management. When working with `PyObject *` variables, careful attention needs to be given to managing reference counts and cleaning up values when no longer needed. The various `Py_DECREF()` calls are doing this.

The recipe is written in a general-purpose manner so that it can be adapted to other file operations, such as writing. For example, to write data, merely obtain the `write()` method of the file-like object, convert data into an appropriate Python object (bytes or Unicode), and invoke the method to have it written to the file.

Finally, although file-like objects often provide other methods (e.g., `readline()`, `read_into()`), it is probably best to just stick with the basic `read()` and `write()` methods for maximal portability. Keeping things as simple as possible is often a good policy for C extensions.

## 15.20. Consuming an Iterable from C

### Problem

You want to write C extension code that consumes items from any iterable object such as a list, tuple, file, or generator.

### Solution

Here is a sample C extension function that shows how to consume the items on an iterable:

```

static PyObject *py_consume_iterable(PyObject *self, PyObject *args) {
    PyObject *obj;
    PyObject *iter;
    PyObject *item;

    if (!PyArg_ParseTuple(args, "O", &obj)) {
        return NULL;
    }
    if ((iter = PyObject_GetIter(obj)) == NULL) {
        return NULL;
    }
    while ((item = PyIter_Next(iter)) != NULL) {
        /* Use item */
        ...
        Py_DECREF(item);
    }
}

```



```

Py_DECREF(iter);
return Py_BuildValue("");
}

```

## Discussion

The code in this recipe mirrors similar code in Python. The `PyObject_GetIter()` call is the same as calling `iter()` to get an iterator. The `PyIter_Next()` function invokes the next method on the iterator returning the next item or `NULL` if there are no more items. Make sure you're careful with memory management—`Py_DECREF()` needs to be called on both the produced items and the iterator object itself to avoid leaking memory.

# 15.21. Diagnosing Segmentation Faults

## Problem

The interpreter violently crashes with a segmentation fault, bus error, access violation, or other fatal error. You would like to get a Python traceback that shows you where your program was running at the point of failure.

## Solution

The `faulthandler` module can be used to help you solve this problem. Include the following code in your program:

```

import faulthandler
faulthandler.enable()

```

Alternatively, run Python with the `-Xfaulthandler` option such as this:

```
bash % python3 -Xfaulthandler program.py
```

Last, but not least, you can set the `PYTHONFAULTHANDLER` environment variable.

With `faulthandler` enabled, fatal errors in C extensions will result in a Python traceback being printed on failures. For example:

```

Fatal Python error: Segmentation fault

Current thread 0x00007fff71106cc0:
  File "example.py", line 6 in foo
  File "example.py", line 10 in bar
  File "example.py", line 14 in spam
  File "example.py", line 19 in <module>
Segmentation fault

```

Although this won't tell you where in the C code things went awry, at least it can tell you how it got there from Python.

## Discussion

The `faulthandler` will show you the stack traceback of the Python code executing at the time of failure. At the very least, this will show you the top-level extension function that was invoked. With the aid of `pdb` or other Python debugger, you can investigate the flow of the Python code leading to the error.

`faulthandler` will not tell you anything about the failure from C. For that, you will need to use a traditional C debugger, such as `gdb`. However, the information from the `faulthandler` traceback may give you a better idea of where to direct your attention.

It should be noted that certain kinds of errors in C may not be easily recoverable. For example, if a C extension trashes the stack or program heap, it may render `faulthandler` inoperable and you'll simply get no output at all (other than a crash). Obviously, your mileage may vary.

---

## APPENDIX A

# Further Reading

There are a large number of books and online resources available for learning and programming Python. However, if like this book, your focus is on the use of Python 3, finding reliable information is made a bit more difficult simply due to the sheer volume of existing material written for earlier Python versions.

In this appendix, we provide a few selected links to material that may be particularly useful in the context of Python 3 programming and the recipes contained in this book. This is by no means an exhaustive list of resources, so you should definitely check to see if new titles or more up-to-date editions of these books have been published.

## Online Resources

*<http://docs.python.org>*

It goes without saying that Python's own online documentation is an excellent resource if you need to delve into the finer details of the language and modules. Just make sure you're looking at the documentation for Python 3 and not earlier versions.

*<http://www.python.org/dev/peps>*

Python Enhancement Proposals (PEPs) are invaluable if you want to understand the motivation for adding new features to the Python language as well as subtle implementation details. This is especially true for some of the more advanced language features. In writing this book, the PEPs were often more useful than the official documentation.

*<http://pyvideo.org>*

This is a large collection of video presentations and tutorials from past PyCon conferences, user group meetings, and more. It can be an invaluable resource for learning about modern Python development. Many of the videos feature Python core developers talking about the new features being added in Python 3.

<http://code.activestate.com/recipes/langs/python>

The ActiveState Python recipes site has long been a resource for finding the solution to thousands of specific programming problems. As of this writing, it contains approximately 300 recipes specific to Python 3. You'll find that many of its recipes either expand upon topics covered in this book or focus on more narrowly defined tasks. As such, it's a good companion.

<http://stackoverflow.com/questions/tagged/python>

Stack Overflow currently has more than 175,000 questions tagged as Python-related (and almost 5000 questions specific to Python 3). Although the quality of the questions and answers varies, there is a lot of good material to be found.

## Books for Learning Python

The following books provide an introduction to Python with a focus on Python 3:

- *Learning Python*, 4th Edition, by Mark Lutz, O'Reilly & Associates (2009).
- *The Quick Python Book*, 2nd Edition, by Vernon Ceder, Manning (2010).
- *Python Programming for the Absolute Beginner*, 3rd Edition, by Michael Dawson, Course Technology PTR (2010).
- *Beginning Python: From Novice to Professional*, 2nd Edition, by Magnus Lie Hetland, Apress (2008).
- *Programming in Python 3*, 2nd Edition, by Mark Summerfield, Addison-Wesley (2010).

## Advanced Books

The following books provide more advanced coverage and include Python 3 topics:

- *Programming Python*, 4th Edition, by Mark Lutz, O'Reilly & Associates (2010).
- *Python Essential Reference*, 4th Edition, by David Beazley, Addison-Wesley (2009).
- *Core Python Applications Programming*, 3rd Edition, by Wesley Chun, Prentice Hall (2012).
- *The Python Standard Library by Example*, by Doug Hellmann, Addison-Wesley (2011).
- *Python 3 Object Oriented Programming*, by Dusty Phillips, Packt Publishing (2010).
- *Porting to Python 3*, by Lennart Regebro, CreateSpace (2011), <http://python3porting.com>.

## Symbols

`!r` formatting code, 244  
`%` (percent) operator, 58, 556  
    `format()` function vs., 88  
`*` (star) operator  
    EBNFs and, 70  
`**kwargs`, 218  
    decorators and, 331  
    enforcing signature on, 364–367  
    help function and, 219  
    wrapped functions and, 353  
`*args`, 217  
    decorators and, 331  
    enforcing signature on, 364–367  
    wrapped functions and, 353  
`+` (plus) operator, 48, 59  
`-O` option (interpreter), 343  
`-OO` option (interpreter), 343  
`-W` (warnings), 584  
    all option, 583  
    error option, 584  
    ignore option, 584  
`.` (dot) operator, 49, 591  
`<` (less than) operator, date comparisons with, 108  
`==` operator, 462  
`?` (question mark) modifier in regular expressions, 48  
`\$` (end-marker) in regular expressions, 44

`_` (single underscore)  
    avoiding clashes with reserved words, 251  
    naming convention and, 250  
`__` (double underscore), naming conventions and, 250

## A

`abc` module, 274  
`abspath()` (`os.path` module), 551  
abstract base classes, 274–276  
    collections module and, 283–286  
    predefined, 276  
abstract syntax tree, 388–392  
abstraction, gratuitous, 593  
`@abstractmethod` decorator (`abc` module), 275  
accepting script via input files, 539  
accessor functions, 238–241  
    adjusting decorators with, 336–339  
`ACCESS_COPY` (`mmap` module), 155  
ACLs, 548  
actor model, 516–520  
`addHandler()` operation (`logging` module), 559  
`add_argument()` method (`ArgumentParser` module), 543, 544  
Advanced Programming in the Unix Environment, 2e (Stevens, Rago), 538  
algorithms, 1–35  
    filtering sequence elements, 26–28

---

*We'd like to hear your suggestions for improving our indexes. Send email to [index@oreilly.com](mailto:index@oreilly.com).*

- finding most frequent item (sequences), 20–21
- limited history, keeping, 5–7
- sorting for largest/smallest N items, 7–8
- unpacking, 1–5
- anonymous functions
  - defining, 224–225
  - variables, capturing, 225–227
- Apache web server, 451
- append() method (ElementTree module), 193
- %apply directive (Swig), 631
- archives
  - creating, 549
  - formats of, 550
  - unpacking, 549
- argparse module, 541, 543, 544
- .args attribute (Exceptions), 579
- ArgumentParser instance, 543
- arguments
  - annotations, 220
  - cleaning up in C extensions, 623
  - multiple-dispatch, implementing with, 376–382
- arrays
  - calculating with large numerical, 97–100
  - high-performance with Cython, 638–642
  - in C extensions, 603
  - large, sending/receiving, 481–483
  - NumPy and, 97–100
  - operating on, with extension functions, 609–612
  - readinto() method and, 152
  - writing to files from, 147
- assertRaises() method (unittest module), 570, 571
- assertRaisesRegex() method (unittest module), 571
- ast module, 78, 311, 388–392
- asynchronous processing, 232–235
- attrgetter() function (operator module), 24
- authentication
  - encryption vs., 463
  - hmac module, 461–463
  - passwords, prompting at runtime, 544
  - requests package and, 439
  - simple, 461–463
  - testing with httpbin.org, 441

## B

- b16decode() function (base64 module), 199
  - binascii functions vs., 198
- b16encode() function (base64 module), 199
  - binascii functions vs., 198
- base64 module, 199
  - binascii module vs., 198
- BaseException class, 577–579
  - except statements and, 576
- BaseRequestHandler (socketserver module), 443
- basicConfig() function (logging module), 556–557
- bin() function, 89–90
- binary data
  - arrays of structures, reading/writing, 199–203
  - encoding with base64 module, 199
  - memory mapping, 153–156
  - nested records, reading, 203–213
  - reading into mutable buffers, 152–153
  - reading/writing, 145–147
  - unpack\_from() method and, 202
  - variable-sized records, reading, 203–213
- binary integers, 89–90
- binascii module, 197–198
- bind() method (Signature objects), 344, 364
- bind\_partial() method (Signature objects), 343
- bit\_length() method (int type), 92
- BNF, 69
- boundscheck() decorator (Cython), 641
- buffer attribute (files), 165
- Buffer Protocol, 147, 609
- BufferedWriter object (io module), 164
- built-ins
  - containers, 593
  - exception class, 578
  - methods, overriding, 256
- byte strings
  - as return value of read(), 146
  - encoding with base64, 199
  - indexing of, 79
  - performing text operations on, 78–81
  - printing, 80
  - writing to text files, 165
- bytes, interpreting as text, 546
- BytesIO() class (io module), 148–149, 570
- bz2 compression format, 550

bz2 module, 149–151  
    compresslevel keyword, 150

## C

C APIs, 612–614

C extension programming, 566, 597  
    APIs, defining/exporting, 612–614  
    acquiring GIL in, 624  
    capsule objects in, 612–614  
    ctypes module and, 599–605  
    Cython, wrapping code in, 632–638  
    file-like objects, reading, 659–662  
    filenames, passing to, 657–658  
    function pointers, converting to callables, 643–644  
    GIL and, 515–516  
    GIL, releasing in, 625  
    iterables from C, consuming, 662–663  
    loading modules, 601  
    modules, building, 607  
    null-terminated strings, passing, 644–648  
    opaque pointers in, 612–614  
    open files, passing to, 658–659  
    operating on arrays with, 609–612  
    Python callables in, 619–624  
    Python/C threads, mixing, 625–626  
    segmentation faults and, 663–664  
    strings, converting to Python objects, 653–654  
    Swig, wrapping with, 627–631  
    Unicode strings, passing, 648–653  
    unknown string type and, 654–657  
    writing, modules, 605–609  
C structures, 147  
calendar module, 108  
\_\_call\_\_() method (metaclasses), 347–349, 356  
callback functions  
    carrying extra state with, 232–235  
    inlining, 235–238  
capsule objects, 612–614  
    C APIs, defining/exporting with, 612–614  
Celery, 457  
center() method (str type), 57  
certificate authorities, 468  
CFFI, 605  
CFUNCTYPE instances (ctypes module), 643  
chain() method (itertools module), 131–132  
chained exceptions, 581, 582

ChainMap class (collections module), 33–35  
    update() method vs., 35  
chdir() function (os module), 537  
check\_output() function (subprocess class), 546  
Chicago, Illinois, 186, 214  
choice() function (random module), 102  
\_\_class\_\_ attribute, 302–304  
class decorators, 279, 355–356  
    data models and, 281  
    mixin classes vs., 281  
    mixins and, 298  
classes, 243–327  
    abstract base classes, 274–276  
    attribute definition order, 359–362  
    caching instances of, 323  
    closures vs., 238–241  
    coding conventions in, 367–370  
    constructors, multiple, 291–292  
    containers, custom, 283–286  
    data models, implementing, 277–283  
    decorators, 279  
    decorators, defining in, 345–347  
    defining decorators as, 347–349  
    delegating attributes, 287–291  
    descriptors and, 264–267  
    extending properties in subclasses, 260–264  
    extending with mixins, 294–299  
    implementing state for objects/machines, 299–305  
    inheritance, implementation of, 258  
    initializing members at definition time, 374–375  
    initializing of data structures in, 270–274  
    lazy attributes and, 267  
    mixins, 260, 294–299  
    private data in, 250–251  
    replacing with functions, 231–232  
    statements, metaclass keyword in, 362–364  
    super() function and, 256–260  
    supporting comparison operations, 321–323  
    type system, implementing, 277–283  
    \_\_init\_\_() method, bypassing, 293–294  
@classmethod decorator, 330  
    decorating class methods with, 350–352  
    \_\_func\_\_ attribute and, 334  
client module (http package), 440  
clock() function (time class), 561  
close() method (generators), 531  
closefd argument (open() function), 166

- closures
  - accessing variables inside, 238–241
  - capturing state with, 233–234
  - classes vs., 238–241
  - converting classes to functions with, 231
  - nonlocal declarations and, 239
- cmath module, 93
- code readability and string templates, 453
- coding conventions, 367–370
- collections module
  - ChainMap class, 33–35
  - Counter class, 20–21
  - defaultdict class, 11
  - deque class, 5–7
  - implementing custom containers with, 283–286
  - namedtuple() function, 30
  - OrderedDict class, 12–13
- combinations() function (itertools module), 126
- combinations\_with\_replacement() function (itertools module), 126
- combining() function (unicodedata module), 51
- command-line options, parsing, 539, 541, 542
- compare\_digest() function (hmac module), 462
- comparison operations, 321–323
- compile() function (ast module), 390
- compile() function (re module), 49
- complex() function, 92
- complex-valued math, 92–94
- compress() function (itertools module), 28
- compressed files, reading/writing, 149–151
  - specifying level of, 150
- concurrency, 485–538
  - actor model and, 516–520
  - coroutines, 524–531
  - event-driven I/O and, 479
  - GIL, 513–516
  - parallel programming, 509–513
  - polling thread queues, 531–534
  - threads, 485–488
  - with generators, 524–531
- Condition object (threading module), 489
- conditionals and NaN values, 95
- config file
  - Python code vs., 554
  - Python source file vs., 553
- ConfigParser module, 552–555
- configuration files, 552
- connect() function (sqlite3 module), 196
- connection (multiprocessing module), 456–458
- constructors, defining multiple, 291–292
- containers, custom, 283–286
  - iterating over items in separate, 131–132
  - \_\_iter\_\_() method and, 114–115
- context managers
  - defining, 384–385
  - use instance as, 540
- context-management protocols, 246–248
- contextlib module, 238
- @contextmanager decorator (contextlib module), 238, 385
- contextmanager module, 248
- cookies, 439
- copy2() function (shutil module), 548
- copying directories/files, 547
- copytree() function (shutil class), 548–549
- coroutines, 524–531
  - capturing state with, 233
  - inlining callback functions with, 235
- Counter objects (collections module), 20–21
- country\_timezones dictionary (pytz module), 112
- cProfile module, 587
- CPU-bound programs, 514
- CPUs, restricting use of, 561
- critical() function (logging module), 556
- cryptography, 103
- CSV files, reading/writing, 175–179
- csv module, 175–179
- ctypes module, 599–605
  - alternatives to, 604
  - memory addresses, wrapping, 643–644
- custom exceptions, 578
- Cython, 604
  - array handling with, 612
  - high-performance arrays with, 638–642
  - memoryviews in, 640
  - releasing the GIL, 636
  - setup.py file, 633, 639
  - wrapping existing code in, 632–638

## D

- daemon processes (Unix), 534–538
- daemonic threads, 486
- data
  - pickle module and, 171–174
  - serializing, 171–174



- transforming/reducing simultaneously, 32–33
- data analysis, 178
- data encapsulation, 250–251
- data encoding
  - CSV files, reading/writing, 175–179
  - for RPCs, 460
  - hexadecimal digits, 197–198
  - JSON, reading/writing, 179–183
  - with base64 module, 199
  - XML, extracting data from, 183–186
- data models, 277–283
- data processing
  - dictionaries, converting to XML, 189–191
  - nested binary records, reading, 203–213
  - parsing large XML files, 186–189
  - relational databases and, 195–197
  - statistics, 214–216
  - summarizing, 214–216
  - variable-sized binary records, reading, 203–213
  - XML namespaces, 193–195
  - XML, parsing/modifying/rewriting, 191–193
- data structures, 1–35, 593
  - cyclic, managing memory in, 317–320
  - dictionaries
    - grouping records by field, 24–26
    - sorting by common keys, 21–23
  - heaps, 7–11
  - initializations of, simplifying, 270–274
  - multidicts, 11–12
- database API (Python), 195–197
  - strings and, 197
- datagrams, 445
- date calculations, 106–107
- datetime module, 104–112
  - databases and, 197
  - date calculations with, 106–107
  - replace() method, 108
  - strptime() method, 109
  - time conversions, 104–105
  - time zones and, 110
  - timedelta object, 104
- dateutil module, 105
- deadlock avoidance, 503
- deadlocks, 500–503
  - dining philosophers problem, 503
  - watchdog timers and, 503
- \_\_debug\_\_ variable, 343
- debug() function (logging module), 556
- debugger, launching, 585
- debugging, 584
- decimal module, 84–86
  - databases and, 197
  - formatting output of, 88
- decode() method (str type), 56
- decorators, 329–356
  - adding function arguments with, 352–354
  - adjustable attributes for, 336–339
  - class methods, applying to, 350–352
  - defining as class, 347–349
  - defining in classes, 345–347
  - forcing type checking with, 341–345
  - multiple-dispatch, implementing with, 381
  - optional arguments for, 339–341
  - patching classes with, 355–356
  - preserving function metadata in, 331–333
  - static methods, applying to, 350–352
  - unwrapping, 333–334
  - with arguments, 334–336
  - wrappers as, 329–331
  - \_\_wrapped\_\_ attribute, 332
- deepcopy() function (copy module), 594
- defaultdict object (collections module), 11
  - groupby() function vs., 26
  - ordinary dictionary vs., 12
- \_\_delattr\_\_() method (delegation), 290
- delegation
  - inheritance vs., 289
  - of attributes, 287–291
- \_\_delete\_\_() method (descriptors), 265
- DeprecationWarning argument (warn() function), 583
- deque class (collections module), 5–7
  - appending, 6
  - popping, 6
- descriptors
  - creating, 264–267
  - data models, implementing, 277–283
  - extending, 263
  - lazy attributes and, 267–270
  - type system, implementing, 277–283
- deserializing data, 293–294
- Design Patterns: Elements of Reusable Object-Oriented Software (Gamma, Helm, Johnson, and Vlissides), 305
- dest argument (add\_argument() method), 544
- detach() method, 164

- dict() function, 29, 594
- dictionaries
  - comparing, 15–16
  - converting to XML, 189–191
  - Counter objects and, 21
  - defaultdict vs., 12
  - grouping records based on field, 24–26
  - JSON support for, 179
  - keeping in order, 12–13
  - multiple values for a single key in, 11–12
  - removing duplicates from, 17
  - sorting, 13–15
  - sorting list of, by common key, 21–23
  - subsets, extracting, 28–29
- dictionary comprehension, 29
- dining philosophers problem (deadlocks), 503
- directories
  - as runnable scripts, 407–408
  - copying, 547
  - moving, 547
  - temporary, 167–170
- directory listings, 158–159
- dis module, 392–395
- discarding vs. filtering iterables, 124
- distributed systems and property calls, 255
- domain sockets (Unix), 457, 470
  - connecting interpreters with, 472
- DOTALL flag (re module), 49
- dropwhile() function (itertools module), 123
- dump() function (pickle module), 171
- dumps() function (json module), 179
  - indent argument, 181
  - sort\_keys argument, 182
- dumps() function (pickle module), 171
- dup2() function (os module), 537

## E

- EBNF, 69
- ElementTree module (xml.etree), 66, 183–186
  - creating XML documents with, 189–191
  - iterparse() function, 188
  - parse() function, 185
  - parse\_and\_remove() function, 189
  - parsing namespaces and, 194
  - parsing/modifying/rewriting files with, 191–193
- empty() method (queue module), 496
- encode() method (str type), 56

- encoding text data, 141–144
  - changing in open files, 163–165
  - passing to C libraries, 648–653
  - pickling and, 174
- encryption vs. authentication, 463
- end argument (print() function), 144
- endswith() method (str type), 38
  - pattern matching with, 42
- \_\_enter\_\_() method (with statements), 246–248
- enumerate() function, 127–128
- environ argument (WSGI), 451
- error messages, terminating program with, 540
- error() function (logging module), 556
- escape() function (html module), 65
- escape() function (xml.sax.saxutils module), 191
- Event object (threading module), 488–491
- event-driven I/O, 475–481
- except statement, 576, 580
  - chained exceptions and, 581
  - raising exceptions caught by, 582
- Exception class, 579
  - handler for, 576
- exceptions
  - catching all, 576
  - creating custom, 578
  - handling multiple, 574
  - raising in response to another exception, 580
  - raising, 582
  - SystemExit, 540
  - testing for, 570
  - with statements and, 247
- exec() function, 386–388
- execve() function (os module), 546
- \_\_exit\_\_() method (with statements), 246–248
- @expectedFailure decorator, 574
- exponential notation, 87
- %extend directive (Swig), 630
- external command
  - executing, 545
  - getting output, 545

## F

- factory functions, 323
- faulthandler module, 663–664
- fdel attribute (properties), 253
- fget attribute (properties), 253
- FieldStorage() class (cgi module), 452
- file descriptor
  - passing between processes, 470–475

- wrapping in object, 166
- file-like objects, reading from C, 659–662
- FileInput class, 540
  - helper methods, 540
- fileinput module (built-in), 539
- FileIO class (io module), 164
- FileNotFoundError exception, 575
- files, 141–174
  - accepting scripts via, 539
  - binary data, reading/writing, 145–147
  - bypassing name encoding, 160–161
  - bytes, writing to text, 165
  - changing encoding in open, 163–165
  - compressed, reading/writing, 149–151
  - copying, 547
  - creating new, 147–148
  - defining modules in multiple, 401–404
  - detach() method, 164
  - directory listings, getting, 158–159
  - finding by name, 550
  - getfilesystemencoding() function (sys module), 160–161
  - headers, unpacking, 205
  - iterating over fixed-sized records in, 151
  - line ending, changing, 144–145
  - manipulating pathnames of, 156–157
  - memory mapping, 153–156
  - moving, 547
  - mutable buffers, reading into, 152–153
  - names, passing to C extensions, 657–658
  - newlines, 142
  - open() function, 141–144
  - open, passing to C extensions, 658–659
  - path module (os), 156–157
  - printing bad filenames for, 161–163
  - printing to, 144
  - reading, 408–409
  - readinto() method of, 147
  - saving objects to, 171–174
  - separator character, changing, 144–145
  - temporary, 167–170
  - testing for existence of, 157–158
  - text data from, 141–144
  - unpacking, 2
  - wrapping descriptors in objects, 166
  - x mode of open() function, 147–148
- files list, 551
- filesystem, byte strings and, 80
- fill() function (textwrap module), 65
- filter() function, 27
- filtering
  - discarding iterables vs., 124
  - normalization of Unicode text and, 51
  - sequence elements, 26–28
  - token streams, 68
- find Unix utility, 551
- find() method (str type)
  - pattern matching with, 42
  - search/replace text and, 46
- findall() method (re module), 43
- finditer() method (re module), 44
- find\_library() function (ctypes.util module), 601
- float()
  - decimal arithmetic and, 84–86
  - format() function (built-in), 87–88
  - j suffix for, 92
- fnmatch module, 40–42
- fnmatch() function (fnmatch module), 40–42
- fnmatchcase() function (fnmatch module), 40–42
- for loops
  - consuming iterators manually vs., 113–114
  - generators and, 115–117
- fork() function (os module), 537
- ForkingTCPServer objects (socketserver module), 442
- ForkingUDPServer objects (socketserver module), 446
- format() function (built-in )
  - and non-decimal integers, 89
  - floats, specifying precision with, 84
  - formatting output with, 244
- format() function (built-in), 87–88
- format() method (str type), 57
  - customizing string, 245–246
  - interpolating variables, 61–64
- \_\_format\_\_() method, 245–246
- format\_map() method (str type), 62
- fpectl module, 95
- fractions module, 96–96
- frame hacking, 373
- from module import \* statement (modules), 398
  - reload() function and, 406
- from module import name, 591
- fromkeys() method (dict type), 55
- from\_bytes() method (int type), 91
- fset attribute (properties), 253

- fsun() function (math module), 86
- full() method (queue module), 496
- functions, 217–241
  - accepting arbitrary arguments for, 217–218
  - accessor functions as attributes of, 238–241
  - anonymous/inline, defining, 224–225
  - argument annotations, 220
  - calling with fewer arguments, 227–230
  - closures, accessing variables inside, 238–241
  - default arguments for, 222–224
  - disassembling, 392–395
  - inlining callbacks, 235–238
  - keyword arguments only, 219–220
  - multiple return values for, 221
  - partial() function and, 227–230
  - pointers, converting to callables, 643–644
  - replacing classes with, 231–232
  - state, carrying, 232–235
  - wrapper layers for, 329–331
  - wrapping with Cython, 632–638
- futures module (concurrent module), 479, 505–509
  - ProcessPoolExecutor class, 509–513
- FutureWarning argument (warn() function), 583

## G

- garbage collection
  - caching instances and, 325
  - cyclic data structures and, 318
- gauss() function (random module), 103
- gdb debugger, 663–664
- General Decimal Arithmetic Specification (IBM), 86
- generator expressions
  - concatenating/combining strings and, 60
  - filtering sequence elements with, 26
  - flattening nested sequences with, 135–136
  - recursive algorithms vs., 311–317
  - transforming/reducing data with, 32–33
- Generator Tricks for Systems Programmers (Beazley), 135
- generator-expression argument, 32
- GeneratorExit exception, 577
- generators, 113–139
  - concurrency with, 524–531
  - creating iterator patterns with, 115–117
  - defining with extra states, 120
  - inlining callback functions with, 235
  - islice() function and, 122–123
  - iterator protocol, implementing, 117–119
  - pipelines, creating with, 132–135
  - search functions and, 6
  - slicing, 122–123
  - unpacking, 2
- GET requests (HTTP), 437
- get() function (webbrowser class), 563
- get() method (ElementTree module), 185
- get() method (queue module), 491, 495
- \_\_get\_\_() method, 347–349
  - descriptors, 265–266
- getattr() function, 305
  - visitor patterns and, 310
- \_\_getattr\_\_() method (delegation), 287, 290
- getboolean() method (ConfigParser module), 554
- getdefaultencoding() function (sys module), 142
- getfilesystemencoding() function (sys module), 160–161
- getframe() (sys module), 63
  - frame hacking with, 373
- \_\_getitem\_\_() method, 23
- getLogger() function (logging module), 558
- getpass module, 544
- getpass() method (getpass module), 545
- getrandbits() function (random module), 103
- getrecursionlimit() function (sys module), 310
- gettempdir() function (tempfile module), 169
- getuser() (getpass module), 544
- get\_archive\_formats() function (shutil module), 550
- get\_data() function (pkgutil module), 409
- get\_terminal\_size() function (os module), 65, 545
- gevent module, 531
- glob module, 42
- Global Interpreter Lock (GIL), 487, 513–516
  - C, calling Python from, 624
  - C/Python threads, mixing, 625–626
  - releasing in C extensions, 625
  - releasing in Cython, 636
  - thread pools and, 508
- grammar rules, for parsers, 69–78
- greenlets, 531
- .group() method (re module), 46
- groupby() function (itertools module), 24–26
- grouping records, in dictionaries, 12
- gzip compression format, 550

gzip module, 149–151  
    compresslevel keyword, 150

## H

hard limit on resources, 562  
hashes, 17  
heappop() method (heapq module), 8  
    priority queues and, 9  
heapq module, 7–8  
    merge() function, 136–137  
heaps, 7–11  
    nlargest()/nsmallest() functions and, 8  
    reversing order of, 10  
help function  
    keyword arguments and, 219  
    metadata for arguments and, 220  
hex() function, 89–90  
hexadecimal digits, encoding, 197–198  
hexadecimal integers, 89–90  
hmac module, 461–463  
HTML entities  
    handling in text, 65–66  
    replacing, 66  
html module, 65  
HTTP services  
    clients, 437–441  
    headers, custom, 438  
    testing client code, 441  
httpbin.org service, 441

## I

I/O, 141–174  
    decoding/encoding functions for, 56  
    event-driven, 475–481  
    iter() function and, 139  
    objects, serializing, 171–174  
    operations on strings, 148–149  
    passing open files and, 659  
    serial ports, communicating with, 170–171  
    thread termination and, 487  
IDEs for Python development, 587  
    \_\_init\_\_() method, data structures and, 273  
if-elif-else blocks, 304  
IGNORECASE flag (re module), 47  
ignore\_patterns() function (shutil module), 548  
ignore\_types argument (isinstance() function), 135

import hooks, 418–420  
    patching modules on import, 428–431  
    sys.path\_hooks variable and, 420  
import statement, 412–428  
ImportError exceptions, 425  
importlib module, 421  
import\_module() function (importlib module), 411, 430  
index-value pairs, iterating over, 127–128  
IndexError exception, 19  
indexing in CSV files, 175–179  
indices() method (slice), 19  
infinite values, 94–95  
info() function (logging module), 556  
inheritance  
    class decorators and, 356  
    class defined decorators and, 347  
    delegation vs., 289  
    implementation of, 258  
.ini file features, 555  
\_\_init\_\_() method (classes), 579  
    bypassing, 293–294  
    coding conventions and, 367–370  
    data structure initialization and, 270–274  
    mixins and, 297  
    multiple constructors and, 291–292  
    optional arguments and, 363  
    super() function and, 256–260  
inline functions  
    callbacks, 235–238  
    defining, 224–225  
input files, 539  
input() function (fileinput), 540  
insert() method (ElementTree module), 193  
inspect() module, 364–367  
instances  
    cached, metaclasses and, 358  
    creation, controlling with metaclasses, 356–359  
    descriptors and, 264–267  
    saving memory when creating, 248–249  
    serializing with JSON, 182  
    WSGI applications as, 452  
int type, 90–92  
int() function, 90  
interface files (Swig), 627  
interpreters, communication between, 456–458  
invalidate\_caches() function (importlib module), 427

- io module, 148–149, 163–165
- ioctl(), 545
- IPv4Address objects (ipaddress module), 448
- is operator, 223
- isinf() function (math module), 94
- isinstance() function, 135
- islice() function (itertools module), 122–123, 124
- isnan() function (math module), 94
- issuing warning messages, 583
- is\_alive() method (Thread class), 486
- itemgetter function (operator module), 22
- items() method (dictionaries), 16
- iter() function
  - fixed size records and, 151
  - while loops vs., 138–139
- \_\_iter\_\_() method (iterators), 114–115
  - generators with extra states and, 121
- iterables
  - custom containers for, 283–286
  - discarding parts of, 123–125
  - from C, consuming, 662–663
  - sorted/merged, iterating over, 136–137
  - unpacking, 1–2
- iterating
  - all possible combinations/permutations, 125–127
  - chain() method (itertools module), 131–132
  - iter() function vs. while loops, 138–139
  - merge() function (heapq module), 136–137
  - over fixed-sized records, 151
  - over index-value pairs, 127–128
  - over separate containers, 131–132
  - over sorted/merged iterables, 136–137
  - reverse, 119
  - zip() function and, 129–130
- iterators, 113–139
  - consuming manually, 113–114
  - creating patterns with generators, 115–117
  - delegating, 114–115
  - islice() function (itertools module), 122–123
  - protocol, implementing, 117–119
  - slicing, 122–123
- iterparse() function (ElementTree module), 188
- iterparse() method (ElementTree module), 194
- itertools module, 123–125, 125–127
  - compress() function, 28
  - groupby() function, 24–26
- iter\_as() function, 211

## J

- join() method (str type), 58–61
  - changing single characters with, 145
- join() method (Thread class), 486
- JSON (JavaScript Object Notation)
  - reading/writing, 179–183
  - types supported by, 179
- json module, 179–183
- just-in-time (JIT) compiling, 590, 594

## K

- key argument (sorted() function), 23
- KeyboardInterrupt exception, 577, 578
- keys() method (dictionaries), 16
- keyword arguments, 218
  - annotations, 220
  - for metaclasses, 362–364
  - functions that only accept, 219–220
  - help function and, 219

## L

- lambda expressions, 24, 225
  - variables, capturing, 225–227
- launching
  - Python debugger, 585
  - web browsers, 562
- lazy imports, 403
- lazy unpacking, 212
- leap years, datetime module and, 104
- libraries, adding logging to, 558
- limited history, 5–7
- linalg subpackage (NumPy module), 101
- linear algebra, 100–101
- list comprehension, 26
- listdir() function (os module), 158–159
  - bad filenames, printing, 161–163
- lists
  - NumPy vs., 99
  - processing and star unpacking, 5
  - unpacking, 2
- ljust() method (str type), 57
- llvmpy extension module, 643
- load() function (pickle module), 171
  - untrusted data and, 172
- LoadLibrary() function (ctypes.cdll module), 601
- loads() function (json module), 179

- loads() function (pickle module), 171
- loadTestsFromModule() method (TestLoader module), 573
- loadTestsFromTestCase() method (TestCase module), 573
- local() method (threading module), 504–505
- locale module, 88
- locals() function and exec() function, 386–388
- Lock objects (threading module), 497–500
- logging
  - adding to libraries, 558
  - adding to simple scripts, 555
  - output of, 557
  - test output to file, 572
- logging module, 557, 559
  - critical(), 556
  - debug(), 556
  - error(), 556
  - info(), 556
  - warning(), 556
- lower() method (str type), 54
- lstrip() method (str type), 53
- lxml module, 195

## M

- MagicMock instances, 568
- makefile() method (socket module), 167
- make\_archive() function (shutil module), 549
- MANIFEST.in file, 434
- manipulating pathnames, 156–157
- map() operation (ProcessPoolExecutor class), 511
- Mapping class (collections module), 283
- mappings
  - class definition order and, 362
  - consolidating multiple, 33–35
- match() method (re module), 43
  - search/replace text and, 46
- math() module, 94
- matrix calculations, 100–101
- matrix object (NumPy module), 100
- max() function, 8, 15
  - attrgetter() function and, 24
  - generator expressions and, 33
  - itemgetter() function, 23
- memory
  - management in cyclic data structures, 317–320
  - mapping, 153–156
  - OrderedDict objects and, 13
  - parsing XML and, 186–189
  - restricting use of, 561
  - wchar\_t strings and, 651
  - \_\_slots\_\_ attribute and, 248–249
- memory mapping, 153–156
- MemoryError exceptions, 562
- memoryview() object (struct module), 153, 206, 213
  - Cython-typed, 640
  - large arrays and, 481–483
- memory\_map() function (mmap module), 154
- merge() function (heapq module), 136–137
- Mersenne Twister algorithm, 103
- meta path importer, 414–418
  - sys.metapath object, 418
- metaclass keyword (class statement), 362–364
- metaclasses, 279
  - capturing attribute definition order, 359–362
  - controlling instance creation, 356–359
  - enforcing coding conventions, 367–370
  - initializing members at definition time, 374–375
  - non-standardized binary files and, 207
  - optional arguments for, 362–364
  - \_\_prepare\_\_() method and, 361
- metaprogramming, 329–395
  - adjustable attributes, 336–339
  - decorators, 329–356
    - decorators with arguments, 334–336
  - dis module, 392–395
  - exec() function vs., 386–388
  - function metadata in decorators, 331–333
  - function wrappers, 329–331
  - repetitive property methods and, 382–384
  - unwrapping decorators, 333–334
- method overloading, 376–382
- methodcaller() function (operator module), 305
- Microsoft Excel, CSV encoding rules of, 177
- Microsoft Visual Studio, 608
- min() function, 8, 15
  - attrgetter() function and, 24
  - generator expressions and, 33
  - itemgetter() function, 23
- \_\_missing\_\_() method (dict module), 62
- mixin classes, 260
  - extending unrelated classes with, 294–299
  - in descriptors of data models, 280
- mkdtemp() function (tempfile module), 169



- mkstemp() function (tempfile module), 169
- mmap module, 153–156
- mock module (unittest module), 565, 570
- module import \* statement, 398–399
- modules, 397–435
  - controlling import of, 398–399
  - from module import \* statement, 398
  - hierarchical packages of, 397–398
  - import hooks, using, 412–428
  - importing with relative names, 399–401
  - importing, using string as name, 411
  - importlib module, 421
  - import\_module() function (importlib module), 411
  - invalidate\_caches() function (importlib module), 427
  - meta path importer, 414–418
  - new\_module() function (imp module), 421
  - objects, creating, 421
  - patching on import, 428–431
  - relative imports vs. absolute names, 400
  - reloading, 406–407
  - remote machines, loading from, 412–428
  - splitting into multiple files, 401–404
  - sys.path, adding directories to, 409–411
  - sys.path\_importer\_cache object, 424
  - virtual environments and, 432–433
  - \_\_all\_\_ variable in, 398–399
  - \_\_init\_\_.py file, 397
  - \_\_main\_\_.py files, 407–408
- monthrange() function (calendar module), 108
- months, finding date ranges of, 107–109
- moving directories/files, 547
- \_\_mro\_\_ attribute, 576
- multidicts, 11–12
- multiple-dispatch, 376–382
- multiprocessing module, 488
  - GIL and, 514
  - passing file descriptors with, 470–475
  - reduction module, 470–475
  - RPCs and, 459
- mutable buffers, reading into, 152–153
- MutableMapping class (collections module), 283
- MutableSequence class (collections module), 283
- MutableSet class (collections module), 283

## N

- named pipes (Windows), 457, 470, 472
- NamedTemporaryFile() function (tempfile module), 169
- namedtuple() function (collections module), 30
- namedtuple() method (collections module)
  - new\_class() function and, 372
  - unpacking binary data and, 203
- NameError exception, 581
- namespace package, checking for, 426
- namespaces (XML), parsing, 193–195
- namespaces, multiple directories in, 404–406
- naming conventions
  - for private data, 250–251
  - \_\_ (double underscore) and, 250
- NaN (not a number) values, 94–95
  - in JSON, 183
  - isnan() function and, 95
- nested sequences, flattening, 135–136
- network programming, 437–483
  - connection (multiprocessing module), 456–458
  - event-driven I/O, 475–481
  - hmac module, 461–463
  - interpreters, communication between, 456–458
  - large arrays, sending/receiving, 481–483
  - passing file descriptors, 470–475
  - remote procedure calls, 454–456, 458–461
  - simple authentication, 461–463
  - socketserver module, 441–444
  - SSL, implementing, 464–470
  - TCP servers, 441–444
  - UDP server, implementing, 445–446
  - UDPServer class, 446
  - XML-RPC, 454–456
- \_\_new\_\_() method (classes)
  - coding conventions and, 367–370
  - optional arguments and, 363
- newlines, 142
- new\_class() function (types module), 370–373
- new\_module() function (imp module), 421
- next() function (iterators), 113
- nlargest() function (heapq module), 7–8
- nonblocking, supporting with queues, 495
- noncapture groups (regular expressions), 49
- nonlocal declarations, 239
  - adjusting decorators with, 336–339



- normalize() function (unicodedata module), 51, 55
- normalize() method (pytz module), 111
- normalizing Unicode, 50–51
  - supported forms in Python, 51
- normpath() (os.path module), 551
- nsmallest() function (heapq module), 7–8
- null-terminated strings, passing to C, 644–648
- NullHandler() class (logging module), 559
- numerical operations, 83–112
  - complex-valued math, 92–94
  - decimal calculations, 84–86
  - formatting for output, 87–88
  - infinity and, 94–95
  - linear algebra calculations, 100–101
  - matrix calculations, 100–101
  - NaNs and, 94–95
  - NumPy and, 97–100
  - on fractions, 96–96
  - packing/unpacking integers from byte string, 90–92
  - random number generators, 102–103
  - rounding, 83–84
  - time, 104–112
  - time conversions, 104–105
  - with binary integers, 89–90
  - with hexadecimal integers, 89–90
  - with octal integers, 89–90
- NumPy module, 97–100, 640
  - complex math and, 93
  - CPU-bound programs and, 514
  - linear algebra and, 100–101
  - matrix calculations and, 100–101
  - unpacking binary data with, 203

## O

- objects, 243–272
  - and context-management protocols, 246–248
  - calling methods on, when named in strings, 305–306
  - creating large numbers of, 248–249
  - defining default arguments and, 224
  - format() function and, 245–246
  - formatting output of, 243–244
  - implementing states for, 299–305
  - iterator protocol, implementing, 117–119
  - JSON dictionary, decoding to, 181
  - memory management, 317–320
  - representing in C, 608
  - serializing, 171–174
  - visitor pattern, implementing, 306–311
  - visitor patterns without recursion, 311–317
  - with statement and, 246–248
- object\_hook (json.loads() function), 181
- object\_pairs\_hook (json.loads() function), 181
- oct() function, 89–90
- octal integers, 89–90
- Olson time zone database, 110
- opaque pointers, 612–614
- open() function, 141–144
  - binary data, reading/writing, 145–147
  - closefd argument, 166
  - file descriptors and, 166
  - on non-Unix systems, 167
  - rb/wb modes of, 145–147
  - rt/wt mode, 141–144
  - x mode of, 147–148
- optimizing your programs, 590
- optparse vs. argparse, 544
- OrderedDict object (collections module), 12–13
  - decoding JSON data into, 181
- os module
  - listdir() function, 158–159
  - path module in, 156–157
- OSError exception, 575–576
- output sent to stdout, 565

## P

- pack() function (structs), 201
- packages, 397–435
  - datafiles, reading, 408–409
  - directories/zip files, running, 407–408
  - distributing, 433–435
  - hierarchical, of modules, 397–398
  - MANIFEST.in file, 434
  - namespaces, multiple directories in, 404–406
  - per-user directory, installing, 431–432
  - submodules, importing with relative names, 399–401
  - sys.path, adding directories to, 409–411
  - third-party options for, 435
  - virtual environments and, 432–433
  - \_\_init\_\_.py file, 397
  - \_\_path\_\_ variable and, 405
- pack\_into() function (struct module), 153
- Pandas package, 178, 214–216
- parallel programming, 509–513

- parse tree, 73
- parse() function (ElementTree module), 185
- parser (html), 66
- parser, recursive descent, 69–78
- parse\_and\_remove() function (ElementTree module), 189
- parsing command-line options, 539, 541
- partial() function (functools module), 227–230
  - defining property methods and, 383
  - fixed size records and, 151
  - optional arguments for decorators and, 341
  - state, carrying with, 234, 235
- Paste, 453
- patch() function (unittest.mock class)
  - testing output with, 565
  - unit tests with, 567
- patching objects, 567
- path module (os), 156–157
  - testing for existing files with, 157–158
- pathnames, manipulating, 156–157
- pattern matching, 42–45
- pattern object (regular expressions), 43
- pdb debugger, 663–664
- per-user directory, 431–432
- performance
  - ast manipulation and, 392
  - attrgetter() function and, 24
  - Cython and, 638–642
  - dictionary comprehension vs. dict() type, 29
  - join() method vs. + operator, 60
  - keeping limited history and, 6
  - lazy attributes and, 267–270
  - nlargest()/nsmallest() functions and, 7
  - NumPy module and, 97–100
  - of byte vs. text strings, 81
  - pattern objects and, 45
  - profiling/timing, 587
  - sanitizing strings and, 56
  - strptime() method and, 110
- perf\_counter() function (time class), 561, 589
- PermissionError exception, 575
- permutations() function (itertools module), 125
- pexpect, 547
- pickle module, 171–174
  - limits on, 172
  - multiprocessing.connection functions and, 457
  - RPCs and, 458–461
- Pipe object (multiprocessing module), 471
- pipelines, creating, 132–135
- pipes
  - accepting script via, 539
  - mimicking, 132–135
- PLY, 69, 76
- polling thread queues, 531–534
- Popen class (subprocess module), 547
- POST method (HTTP), 438
- pprint() function (pprint module), 180
- predefined abstract base classes, 276
- prefix variable, 554, 555
- \_\_prepare\_\_() method (classes), 361
  - new\_class() function and, 372
  - optional arguments and, 363
- prepare\_class() function (types module), 373
- print() function, 243, 542, 586
  - end argument, 144
  - line ending, changing, 144–145
  - redirecting to files, 144
  - separator character, changing, 144–145
- private data/methods
  - caching instances within, 326
  - naming conventions for, 250–251
- probability distributions (random module), 103
- process pools, GIL and, 514, 516
- ProcessPoolExecutor class (futures module), 509–513
- process\_time() function (time class), 561, 589
- program crashes, debugging, 584
- program profiling, 587
- property attributes (classes)
  - delegating, 287–291
  - extending in subclasses, 260–264
  - repetitive definitions for, 382–384
- @property decorator, 330, 346
- proxies, delegating attributes with, 288, 290
- public-facing services, 457
- put() method (queue module), 491, 495
- pwd module, 544
- PyArg\_ParseTuple() function (C extensions), 609
  - converting string encoding with, 650
- PyBuffer\_GetBuffer() method (Py\_buffer object), 611
- PyBuffer\_Release() method (Py\_buffer object), 612
- PyCallable\_Check() function (C extensions), 622

- PyCapsule\_GetPointer() function (C extensions), 614
- PyCapsule\_Import() function (C extensions), 618
- PyCapsule\_New() function (C extensions), 613
- PyCapsule\_SetDestructor() function (C extensions), 614
- PyErr\_Occurred() function (C extensions), 623
- PyFile\_FromFd() function (C extensions), 658–659
- PyFloat\_AsDouble() function (C extensions), 623
- PyFloat\_Check() function (C extensions), 623
- PyGILState\_Ensure() function (C extensions), 624, 626
- PyGILState\_Release() function (C extensions), 624, 626
- PyIter\_Next() function (C extensions), 662–663
- PyObject data type (C extension)
  - filenames, passing with, 657–658
- PyObject\_BuildValue function (C extensions), 623
- PyObject\_Call() function (C extensions), 622
  - file-like objects and, 661
- PyObject\_GetIter() function (C extensions), 662–663
- PyParsing, 69, 76
- PyPy, 514
- PySequence\_Length() function (C extensions), 661
- pySerial package, 170–171
- Python for Data Analysis (McKinney), 216
- Python Package Index, 434
- Python syntax vs. JSON, 180
- PYTHONPATH environment variable, 409
- pytz module, 110–112
- PyUnicode\_FromWideChar() function (C extensions)
  - C strings, converting to Python objects, 653–654
- pyvenv command, 432–433
- Py\_BEGIN\_ALLOW\_THREADS (Py objects), 625
- Py\_BuildValue() function (C extensions), 609
  - C strings, converting to Python objects, 653–654
- Py\_DECREF() function (C extensions), 623
- Py\_END\_ALLOW\_THREADS (Py objects), 625
- Py\_XDECREF() function (C extensions), 623

## Q

- qsize() method (queue module), 496
- queue module, 491–496
- queues
  - bounding, 495
  - structures, deque() and, 6
- quote() function (shlex class), 546

## R

- race conditions, avoiding, 497–500
- raise from statement, 580, 582
  - None modifier, 581
- raise statement, 580, 582
- randint() function (random module), 102
- random module, 102–103
- random() function (random module), 103
- re module
  - compile() function, 49
  - DOTALL flag, 49
  - IGNORECASE flag, 47
  - pattern matching and, 42–45
  - patterns, naming, 67
  - scanner() method, 67
  - search/replace text and, 45–46
  - split() method, 37–38
  - sub() function, 54
  - Unicode text and, 52–53
- read() function, 139
  - file-like C objects and, 659–662
- readability, naming slices for, 18–19
- readinto() method (files), 147
  - reading into mutable buffers with, 152–153
- recursion, 5
  - getrecursionlimit() function (sys module), 310
- recursive descent parsers, 69–78
  - limitations on, 76
- recvfrom() method (socket module), 445
- recv\_handle() function (reduction module), 470–475
- recv\_into() function (socket module), 153, 483
- redirection, 539
- regex module, 53
- register() function (atexit module), 538
- register\_function() method (XML-RPC), 455
- regular expressions
  - \* (star) operator and, 48
  - ? (question mark) modifier and, 48

- greedy vs. nongreedy, 47–48
- matching multiple lines, 48–49
- newline support in, 48–49
- on byte strings, 79
- order of tokens in, 68
- pattern matching and, 42–45
- `re.split()` method and, 38
- regex module for, 53
- shortest match with, 47–48
- stripping characters with, 54
- Unicode and, 52–53
- relational databases, 195–197
  - connecting to, 196
  - cursors, creating, 196
- relative imports vs. absolute names (modules), 400
- `relativedelta()` function (dateutil module), 105
- `reload()` function (imp module), 406–407, 431
- remote machines
  - loading modules from, 412–428
  - XML-RPC, 454–456
- remote procedure calls (RPC), 458–461
  - exception handling with, 461
- `remove()` method (ElementTree module), 193
- `replace()` method (date module), 108
- `replace()` method (datetime module), 108
- `_replace()` method (namedtuple object), 31
- `replace()` method (str type), 54, 54
  - performance and, 56
  - search/replace text and, 45
- `repr()` function (built-in), 243
- request module (urllib module), 438–441
  - client package vs., 440
- requests package (urllib module)
  - return values of, 439
- raising exceptions, 582
- reserved words, clashing with, 251
- resource forks, 548
- resource management, 248
- resource module, 561
- RuntimeWarning argument (warn() function), 583
- REST-base interface, 449–453
  - testing, 450
- restricting CPU time, 561
- `result()` method (ProcessPoolExecutor class), 512
- reverse iteration, 119–120
- `reversed()` function, 119–120
- `rjust()` method (str type), 57
- RLock objects (threading module), 498
- `round()` function, 83–84
- rounding numbers, 83–84
- RSS feeds, parsing, 183
- `rstrip()` method (str type), 53
- RTS-CTS handshaking, 170
- RuntimeWarning argument (warn() function), 583

## S

- `sample()` function (random module), 102
- sanitizing text, 54–57
- `scanner()` method (re module), 67
- search
  - for shortest match with regular expressions, 47–48
  - matching multiple lines, 48–49
  - `nlargest()/nsmallest()` functions (heapq module), 7–8
  - noncapture groups, 49
  - normalization of Unicode text and, 51
  - visitor pattern and, 311
- search/replace text, 45–46
  - case insensitive, 46–47
- security
  - import statement and, 412
  - pickle and, 172
  - RPCs and, 460
  - SSL certificates and, 466
- `seed()` function (random module), 103
- segmentation faults, 663–664
- `select()` function (event driven I/O), 476
- self-signed certificates (SSL), 468
- Semaphore objects (threading module), 490, 498
- `send()` function (socket module), 483
- `send()` method (generators), 316
- `sendmsg()` method (socket module), 473
- `sendto()` method (socket module), 445
- `send_handle()` function (reduction module), 470–475
- Sequence class (collections module), 283
- sequences
  - filtering elements of, 26–28
  - flattening nested, 135–136
  - iterating over multiple, simultaneously, 129–130
  - mapping names to elements of, 29–32

- most frequently occurring item in, 20–21
  - removing duplicates from, 17–18
  - unpacking, 1–2
- serial ports, communicating with, 170–171
- serve\_forever() method (XML-RPC), 455
- Set class (collections module), 283
- \_\_set\_\_() method (descriptors), 265
  - in data models, 280
- setattr() method, 294
- \_\_setattr\_\_() method (delegation), 290
- setrlimit() function, 562
- setuid() function (os module), 537
- setup.py file, 633, 639
  - distributing packages and, 434
- set\_trace() function (pdb module), 586–587
- shell, 546
- shell scripts, 539
- shelling out, 550
- shuffle() function (random module), 102
- shutil module
  - archives and, 549
  - copying files/directories with, 547
- sig module, 343–345
- signature objects, 364–367
- signature() function (inspect), 343
- SIGXCPU signal, 562
- simple scripts, 555
- simplefilter() function (warnings class), 584
- SimpleXMLRPCServer (XML-RPC), 456
- site-packages directories, 410
  - virtual environments vs., 432–433
- skip() decorator, 574
- skipIf() function (unittest module), 574
- skipping test failures, 573
- skipUnless() function (unittest module), 574
- slice() object, 19
- slices, naming, 18–19
- \_\_slots\_\_ attribute
  - classes with, 32
  - memory management and, 249
- socket module, 444
  - ipaddress module and, 448
  - sending datagrams with, 445
- socketpair() function (Unix), 532
- sockets
  - large arrays, sending/receiving, 483
  - setting options on, 443
  - threads and, 532
- socketserver module
  - implementing TCP servers with, 441–444
  - UDP server, implementing, 445–446
- sorted() function
  - and objects without comparison support, 23–24
  - itemgetter() function, 22
- sorting
  - dictionaries, 13–15
  - dictionaries by common key, 21–23
  - finding largest/smallest N items, 7–8
  - groupby() function and, 25
  - itemgetter function (operator module), 22
  - objects without comparison support, 23–24
- sort\_keys argument (json.dumps() function), 182
- source file vs. config file, 553
- special characters, escaping, 66
- split() method (re object), 37–38
- split() method (str type), 37–38
- SQLAlchemy, 197
- sqlite3 module, 195
- sqrt() function (math module), 592
- ssh session, 547
- SSL, 464–470
  - certificate authorities, 468
  - self-signed certificates, 468
- ssl module, 464–470
  - random module vs., 103
- Stackless Python, 531
- stack\_size() function (threading module), 509
- star expressions
  - discarding values from, 4
  - unpacking iterables and, 3
- start attribute (slice), 19
- start() method (Thread class), 486
- startswith() method (str type), 38
  - pattern matching with, 42
- start\_response argument (WSGI), 452
- state
  - capturing with closures, 233–234
  - carrying with callback functions, 232–235
  - implementing for objects/machines, 299–305
  - of mixins, 297
  - thread-specific, storing, 504–505
- states, generators with extra, 120–121
- static methods, applying decorators to, 350–352

- @staticmethod decorator, 330
  - decorating class methods with, 350–352
  - \_\_func\_\_ attribute and, 334
- stdout object (sys), changing encoding on, 163
- step attribute (slice), 19
- stop attribute (slice), 19
- StopIteration exception, 113
- str type
  - converting to datetime objects, 109–110
  - decode() method, 56
  - encode() method, 56
  - endswith() method, 38
  - format() function, 57
  - join() method, 58–61
  - lower() method, 54
  - pattern matching with, 42
  - replace() method, 54
  - sanitizing text with, 54–57
  - split() method, 37–38
  - startswith() method, 38
  - stripping unwanted characters from, 53–54
  - translate() method, 55
  - upper() method, 54
- str() function, 243
- StreamRequestHandler (socketserver module), 442–444
- string module, 246
- string templates and code readability, 453
- StringIO object (io module), 566
- StringIO() object (io module), 148–149
- strings, 37–81
  - aligning, 57–58
  - bad encoding in C/Python extensions, 654–657
  - C, converting to Python objects, 653–654
  - calling object methods when named in, 305–306
  - combining, 58–61
  - concatenating, 58–61
  - converting to dates/times, 109–110
  - database API and, 197
  - I/O operations, performing on, 148–149
  - interpolating variables in, 61–64
  - matching start/end text of, 38–40
  - null-terminated, passing to C, 644–648
  - splitting on delimiters, 37–38
  - stripping unwanted characters from, 53–54
  - Unicode, passing to C modules, 648–653
  - unpacking, 2
- strip() method (str type), 53–54
- strptime() method (datetime module), 109
- struct module, 199–203
  - nested binary records, reading, 203–213
  - packing/unpacking integers from byte strings and, 91
  - structure codes for, 201
  - variable-sized binary records, reading, 203–213
- structures (data type), 612–614
- sub() function (re module), 54, 63
  - search/replace text and, 45
- submit() operation (ProcessPoolExecutor class), 512
- subn() method (re module), 46
- subprocess module, 547
- substitution callback functions, 46
- super() function, 256–260
  - class decorators and, 356
  - coding conventions and, 370
  - extending properties in subclasses and, 262
  - mixin classes and, 298
- surrogate encoding, 654–657
- Swig, 604, 627–631
  - headers and, 630
- symbolic links, 548
  - broken, 549
- SyntaxWarning argument (warn() function), 583
- sys.arg value, 543
- sys.argv (comma and line arguments), 544
- sys.metapath object, 418, 430
  - extending import operations and, 422
- sys.modules dictionary, 421
- sys.path
  - adding directories to, 409–411
  - site-packages directories, 410
- sys.path\_hooks variable (importers), 420
- sys.path\_importer\_cache object, 424
- sys.stderr, 540
- sys.stdout, 565–566
- system-exiting exceptions, 578
- SystemExit exception, 540, 577–580

## T

- tag attribute (ElementTree module), 185
- tarfile compression format, 550
- TCP servers, 441–444
  - event-driven I/O implementation, 477

- tempfile module, 167–170
- temporary files, 167–170
- TemporaryDirectory() method (tempfile module), 169
- TemporaryFile (tempfile module), 168
- terminal, finding size of, 65, 545
- test failures, 573
- TestCase classes (TestLoader module), 573
- testing
  - output sent to stdout, 565
  - output, logging to file, 572
  - unit tests for exceptions, 570
- TestLoader class (unittest module), 573
- text attribute (ElementTree module), 185
- text data
  - encoding, 141–144
  - reading/writing, 141–144
- text manipulation, 37–81
  - aligning strings, 57–58
  - case insensitive search/replace, 46–47
  - combining/concatenating, 58–61
  - HTML entities, handling in text, 65–66
  - interpolating variables, 61–64
  - matching start/end of strings, 38–40
  - of Unicode with regular expressions, 52–53
  - on byte strings, 78–81
  - parsers, implementing, 69–78
  - pattern matching, 42–45
  - reformatting into columns, 64–65
  - sanitizing, 54–57
  - search/replace, 45–46
  - stripping unwanted characters, 53–54
  - tokenizing, 66–69
  - wildcard matching, 40–42
  - XML entities, handling in text, 65–66
- TextIOWrapper object (io module), 163–165
- textwrap module, 64–65
- Thread class (threading module), 485–488
- thread pools
  - GIL and, 508
  - queues and, 506
- threading module, 488
  - Condition object, 489
  - Event object, 488–491
  - local() method, 504–505
  - Lock objects, 497–500
  - Semaphore objects, 490
  - stack\_size() function, 509
- ThreadingMixIn class (socketserver module), 297
- ThreadingTCPServer objects (socketserver module), 442
- ThreadingUDPServer objects (socketserver module), 446
- ThreadPoolExecutor class (futures module), 505–509
- threads
  - actor model and, 516–520
  - C/Python, mixing, 625–626
  - communication between, 491–496
  - creating/destroying, 485–488
  - daemonic, 486
  - deadlocks between, 500–503
  - generators as alternative to, 524–531
  - locking critical sections, 497–500
  - nonblocking, supporting with queues, 495
  - polling multiple queues, 531–534
  - pools of, 505–509
  - priority queues and, 11
  - queue module and, 491–496
  - race conditions, avoiding, 497–500
  - status of, finding, 488–491
  - storing state of, 504–505
  - timeouts, supporting with queues, 495
- throw() method (generators), 531
- time command, 587
- time module, 590
- time zones, 110–112
  - country\_timezones dictionary (pytz module), 112
  - UTC time, 111
- time() function (time class), 561
- time, operations on, 104–112
  - converting strings for, 109–110
  - date calculations, 106–107
  - date ranges, finding, 107–109
  - pytz module, 110–112
  - time conversions, 104–105
  - time zones, manipulating, 110–112
  - UTC time, 111
- timedelta object (datetime module), 104, 107–109
- timeit module, 589, 590
- timeouts, supporting with queues, 495
- Timer class, 561
- tokenizing and DOTALL flag (re module), 49
- tokens streams, filtering, 68



- tostring() function (ElementTree module), 190
- total\_ordering decorator (functools module), 321–323
- to\_bytes() method (int module), 91
- traceback, segmentation faults and, 663–664
- traceback.print\_stack() function, 586
- translate() method (str type), 54, 55
  - numerical output and, 88
  - performance and, 56
- transmitting data, 155
- tree structures, memory management of, 317–320
- tree traversal, 311, 314
- TTYs, 545, 547
- tuples
  - and endswith()/startswith() methods, 39
  - as return values, 221
  - relational databases and, 195–197
  - unpacking, 2
- Twisted package, 238
- type checking (data)
  - abstract base classes and, 274–276
  - forcing with decorator functions, 341–345
- type systems, 277–283
- %typemap directive (Swig), 631
- types module, 370–373

## U

- UDP server, 445–446
  - event-driven I/O implementation, 476
- UDPServer class, 446
- umask() function (os module), 537
- unescape() function (xml.sax.saxutils module), 191
- Unicode, 50–53
  - bad encoding in C/Python extensions, 654–657
    - IGNORECASE flag and matching, 47
    - regular expressions and, 52–53
    - strings, passing to C modules, 648–653
- unicodedata module, 55
- uniform() function (random module), 103
- unittest module, 565, 572–573
- Unix, 548
  - commands, 548
  - find utility, 551
- unpack() function (struct module), 201
  - binary data and, 205

- unpacking
  - archives, 549
  - discarding values while, 2
  - enumerate() function and, 128
  - integers from byte string, 90–92
  - iterables into separate variables, 1–2
  - iterables of arbitrary length, 3–5
  - sequences into separate variables, 1–2
  - star expressions and, 3
- unpack\_archive() function (shutil module), 549
- unpack\_from() method (struct module), 202
- uploading using requests module, 440
- upper() method (str type), 54
- urllib module, 413, 437–441
- urlopen() function (urllib module), 569–570
  - import statements vs., 413
  - sending query parameters with, 438
- UserWarning argument (warn() function), 583
- UTC time, 111

## V

- validation of data, abstract base classes and, 274–276
- ValueError, 570
  - unpacking and, 2
- values() method (dictionaries), 14, 16
- variables
  - interpolating, in strings, 61–64
  - understand locality of, 592
- vars() method, 62
- virtual environments, 432–433
- visitor pattern
  - implementing with recursion, 306–311
  - implementing without recursion, 311–317

## W

- walk() function (os module), 550
- warn() function (warning class), 583
- warning messages, issuing, 583
- warning module, 584
- warning() function (logging module), 556
- watchdog timers, 503
- wchar\_t \* declarations (C), 648–653
  - converting to Python objects, 653
- weak references and caching instances, 325
- weakref module, 317, 320
- web browsers, launching, 562



- web programming, 437–483
  - HTTP service clients, 437–441
  - REST-base interface, 449–453
  - urllib module and, 437–441
- webbrowser module, 563
- WebOb, 453
- while loops vs. `iter()` function, 138–139
- wildcard matching, strings, 40–42
- Windows, 548
  - C extension modules and, 608
- with nogil: statement (Cython), 636
- with statement, 246–248, 561, 566
  - contextmanager decorator and, 385
  - Lock objects and, 497–500
  - Semaphore objects and, 499
- wraparound() decorator (Cython), 641
- `__wrapped__` attribute, 332
- `@wraps` decorator (functools)
  - function metadata and, 331–333
  - unwrapping, 333–334
- WSGI standard, 449–453
  - return value for apps based on, 452

## X

- XML entities
  - handling in text, 65–66
  - replacing, 66
- XML files
  - dictionaries, converting to, 189–191

- extracting data from, 183–186
- modifying, 191–193
- parsing, 191–193
  - incrementally, 186–189
  - with namespaces, 193–195
- rewriting, 191–193
- tags, specifying, 185
- XML-RPC, 454–456
  - adding SSL to, 466
  - data types handled by, 455
- `xml.etree.ElementTree` module (see `ElementTree` module)
- `xml.sax.saxutils` module, 191
- `XMLNamespaces` class, 194

## Y

- yield from statement, 135–136
- yield statement, 60, 134
  - behavior in generators, 315
  - concurrency implementations and, 524–531
  - generators and, 116
  - search functions and, 6

## Z

- ZeroMQ, 457
- zip files as runnable scripts, 407–408
- `zip()` method (dictionaries), 13–15, 129–130
- zipfile compression format, 550

## About the Authors

---

**David Beazley** is an independent software developer and book author living in the city of Chicago. He primarily works on programming tools, providing custom software development, and teaching practical programming courses for software developers, scientists, and engineers. He is best known for his work with the Python programming language, for which he has created several open source packages (e.g., Swig and PLY) and authored the acclaimed *Python Essential Reference*. He also has significant experience with systems programming in C, C++, and assembly language.

**Brian K. Jones** is a system administrator in the department of computer science at Princeton University.

## Colophon

---

The animal on the cover of *Python Cookbook*, Third Edition is a springhaas (*Pedetes capensis*), also known as a spring hare. Springhaas are not hares at all, but rather the only member of the family *Pedetidae* in the order Rodentia. They are not marsupials, but they are vaguely kangaroo-like, with small front legs, powerful hind legs designed for hopping, jumping, and leaping, and long, strong, bushy (but not prehensile) tails used for balance and as a brace when sitting. They grow to be about 14 to 18 inches long, with tails as long as their bodies, and can weigh approximately eight pounds. Springhaas have rich, glossy, tawny, or golden-reddish coats with long, soft fur and white underbellies. Their heads are disproportionately large, and they have long ears (with a flap of skin at the base they can close to prevent sand from getting inside while they are digging) and large, dark brown eyes.

Springhaas mate throughout the year and have a gestation period of about 78 to 82 days. Females generally give birth to only one baby (which stays with its mother until it is approximately seven weeks old) per litter but have three or four litters each year. Babies are born with teeth and are fully furred, with their eyes closed and ears open.

Springhaas are terrestrial and well-adapted for digging, and they tend to spend their days in the small networks of their burrows and tunnels. They are nocturnal and primarily herbivorous, feeding on bulbs, roots, grains, and occasionally insects. While they are foraging, they move about on all fours, but they are able to move 10 to 25 feet in a single horizontal leap and are capable of quick getaways when frightened. Although they are often seen foraging in groups in the wild, they do not form an organized social unit and usually nest alone or in breeding pairs. Springhaas can live up to 15 years in captivity. They are found in Zaire, Kenya, and South Africa, in dry, desert, or semiarid areas, and they are a favorite and important food source in South Africa.

The cover image is from *Animal Creation: Mammalia*. The cover font is Adobe ITC Garamond. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.

O'REILLY

ALPHA

# Fluent Python

Clear, Concise, and Effective Programming



Early  
Release

Now &  
Available

Luciano Ramalho

## 1. I. Prologue

### 2. 1. The Python Data Model

- a. What's new in this chapter
- b. A Pythonic Card Deck
- c. How Special Methods Are Used
  - i. Emulating Numeric Types
  - ii. String Representation
  - iii. Arithmetic Operators
  - iv. Boolean Value of a Custom Type
  - v. Collection API
- d. Overview of Special Methods
- e. Why len Is Not a Method
- f. Chapter Summary
- g. Further Reading

## 3. II. Data Structures

### 4. 2. An Array of Sequences

- a. What's new in this chapter
- b. Overview of Built-In Sequences
- c. List Comprehensions and Generator Expressions
  - i. List Comprehensions and Readability
  - ii. Listcomps Versus map and filter
  - iii. Cartesian Products

#### iv. Generator Expressions

#### d. Tuples Are Not Just Immutable Lists

##### i. Tuples as Records

##### ii. Unpacking

##### iii. Nested Tuple Unpacking

##### iv. Tuples as Immutable Lists

##### v. Tuple versus list methods

#### e. Slicing

##### i. Why Slices and Range Exclude the Last Item

##### ii. Slice Objects

##### iii. Multidimensional Slicing and Ellipsis

##### iv. Assigning to Slices

#### f. Using + and \* with Sequences

##### i. Building Lists of Lists

#### g. Augmented Assignment with Sequences

##### i. A += Assignment Puzzler

#### h. list.sort and the sorted Built-In Function

##### i. Managing Ordered Sequences with bisect

##### i. Searching with bisect

##### ii. Inserting with bisect.insort

#### j. When a List Is Not the Answer

##### i. Arrays

- ii. Memory Views
- iii. NumPy and SciPy
- iv. Deque and Other Queues

k. Chapter Summary

l. Further Reading

5. 3. Dictionaries and Sets

- a. What's new in this chapter
- b. Standard API of Mapping Types
- c. dict Comprehensions
- d. Overview of Common Mapping Methods
  - i. Handling Missing Keys with.setdefault
- e. Mappings with Flexible Key Lookup
  - i. defaultdict: Another Take on Missing Keys
  - ii. The \_\_missing\_\_ Method
- f. Variations of dict
- g. Building custom mappings
  - i. Subclassing UserDict
- h. Immutable Mappings
- i. Dictionary views
- j. Set Theory
  - i. Set Literals
  - ii. Set Comprehensions
  - iii. Set Operations

iv. Set operations on dict views

k. Internals of sets and dicts

i. A Performance Experiment

ii. Set hash tables under the hood

iii. The hash table algorithm

iv. Hash table usage in dict

v. Key-sharing dictionary

vi. Practical Consequences of How dict Works

l. Chapter Summary

m. Further Reading

6. 4. Text versus Bytes

a. What's new in this chapter

b. Character Issues

c. Byte Essentials

d. Basic Encoders/Decoders

e. Understanding Encode/Decode Problems

i. Coping with UnicodeEncodeError

ii. Coping with UnicodeDecodeError

iii. SyntaxError When Loading Modules with  
Unexpected Encoding

iv. How to Discover the Encoding of a Byte  
Sequence

v. BOM: A Useful Gremlin

f. Handling Text Files

- i. [Encoding Defaults: A Madhouse](#)
  - g. [Normalizing Unicode for Saner Comparisons](#)
    - i. [Case Folding](#)
    - ii. [Utility Functions for Normalized Text Matching](#)
    - iii. [Extreme “Normalization”: Taking Out Diacritics](#)
  - h. [Sorting Unicode Text](#)
    - i. [Sorting with the Unicode Collation Algorithm](#)
  - i. [The Unicode Database](#)
    - i. [Finding characters by name](#)
    - ii. [Numeric meaning of characters](#)
  - j. [Dual-Mode str and bytes APIs](#)
    - i. [str Versus bytes in Regular Expressions](#)
    - ii. [str Versus bytes in os Functions](#)
  - k. [Multi-character emojis](#)
    - i. [Country flags](#)
    - ii. [Skin tones](#)
    - iii. [Rainbow flag and other ZWJ sequences](#)
  - l. [Chapter Summary](#)
  - m. [Further Reading](#)
- 7. [5. Record-like data structures](#)
  - a. [What’s new in this chapter](#)



- b. Overview of data class builders
  - i. Main features
- c. Classic Named Tuples
- d. Typed Named Tuples
- e. Type hints 101
  - i. No runtime effect
  - ii. Variable annotation Syntax
  - iii. The meaning of variable annotations
- f. More about @dataclass
  - i. Field options
  - ii. Post-init processing
  - iii. Typed class attributes
  - iv. Initialization variables that are not fields
  - v. @dataclass Example: Dublin Core Resource Record
- g. Data class as a code smell
  - i. Data class as scaffolding
  - ii. Data class as intermediate representation
- h. Parsing binary records with struct
  - i. Structs and Memory Views
  - ii. Should we use struct?
- i. Chapter Summary
- j. Further Reading

## 8. 6. Object References, Mutability, and Recycling

- a. What's new in this chapter
- b. Variables Are Not Boxes
- c. Identity, Equality, and Aliases
  - i. Choosing Between == and is
  - ii. The Relative Immutability of Tuples
- d. Copies Are Shallow by Default
  - i. Deep and Shallow Copies of Arbitrary Objects
- e. Function Parameters as References
  - i. Mutable Types as Parameter Defaults: Bad Idea
  - ii. Defensive Programming with Mutable Parameters
- f. del and Garbage Collection
- g. Weak References
  - i. The WeakValueDictionary Skit
  - ii. Limitations of Weak References
- h. Tricks Python Plays with Immutables
- i. Chapter Summary
- j. Further Reading

# Part I. Prologue

---

# Chapter 1. The Python Data Model

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at [fluentpython2e@ramalho.org](mailto:fluentpython2e@ramalho.org).

*Guido’s sense of the aesthetics of language design is amazing. I’ve met many fine language designers who could build theoretically beautiful languages that no one would ever use, but Guido is one of those rare people who can build a language that is just slightly less theoretically beautiful but thereby is a joy to write programs in.<sup>1</sup>*

—Jim Hugunin, Creator of Jython, cocreator of  
AspectJ, architect of the .Net DLR

One of the best qualities of Python is its consistency. After working with Python for a while, you are able to start making informed, correct guesses about features that are new to you.

However, if you learned another object-oriented language before Python, you may find it strange to use `len(collection)` instead of `collection.len()`. This apparent oddity is the tip of an iceberg that, when properly understood, is the key to everything we call *Pythonic*. The iceberg is called the Python Data Model, and it describes the API

that you can use to make your own objects play well with the most idiomatic language features.

You can think of the data model as a description of Python as a framework. It formalizes the interfaces of the building blocks of the language itself, such as sequences, iterators, functions, coroutines, classes, context managers, and so on.

When using a framework, we spend a lot of time coding methods that are called by the framework. The same happens when we leverage the Python Data Model. The Python interpreter invokes special methods to perform basic object operations, often triggered by special syntax. The special method names are always written with leading and trailing double underscores (i.e., `__getitem__`). For example, the syntax `obj[key]` is supported by the `__getitem__` special method. In order to evaluate `my_collection[key]`, the interpreter calls `my_collection.__getitem__(key)`.

The special method names allow your objects to implement, support, and interact with fundamental language constructs such as:

- Collections;
- Attribute access;
- Iteration (including asynchronous iteration using `async for`);
- Operator overloading;
- Function and method invocation;
- String representation and formatting;
- Asynchronous programming using `await`;
- Object creation and destruction;

- Managed contexts (including asynchronous context managers using `async with`).

## MAGIC AND DUNDER

The term *magic method* is slang for special method, but how do we talk about a specific method like `__getitem__`? I learned to say “dunder-getitem” from author and teacher Steve Holden. Most experienced Pythonistas understand that shortcut. As a result, the special methods are also known as *dunder methods*.

## What’s new in this chapter

This chapter had few changes from the first edition because it is an introduction to the Python Data Model, which is quite stable. The most significant changes are:

- Special methods supporting asynchronous programming and other new features, added to the tables in “[Overview of Special Methods](#)”.
- [Figure 1-2](#) showing the use of special methods in “[Collection API](#)”, including the `collections.abc.Collection` abstract base class introduced in Python 3.6.

Also, here and throughout this 2nd edition I’ve adopted the *f-string* syntax introduced in Python 3.6, which is more readable and convenient than the older string formatting notations: the `str.format()` method and the `%` operator. The most common reason to use `my_fmt.format()` is to build the template `my_fmt` at runtime. That is a real need, but doesn’t happen very often.

## A Pythonic Card Deck

Example 1-1 is very simple, but it demonstrates the power of implementing just two special methods, `__getitem__` and `__len__`.

*Example 1-1. A deck as a sequence of playing cards*

---

```
import collections

Card = collections.namedtuple('Card', ['rank', 'suit'])

class FrenchDeck:
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        self._cards = [Card(rank, suit) for suit in self.suits
                        for rank in self.ranks]

    def __len__(self):
        return len(self._cards)

    def __getitem__(self, position):
        return self._cards[position]
```

The first thing to note is the use of `collections.namedtuple` to construct a simple class to represent individual cards. Since Python 2.6, we can use `namedtuple` to build classes of objects that are just bundles of attributes with no custom methods, like a database record. In the example, we use it to provide a nice representation for the cards in the deck, as shown in the console session:

```
>>> beer_card = Card('7', 'diamonds')
>>> beer_card
Card(rank='7', suit='diamonds')
```

But the point of this example is the `FrenchDeck` class. It's short, but it packs a punch. First, like any standard Python collection, a deck responds to the `len()` function by returning the number of cards in it:

```
>>> deck = FrenchDeck()
>>> len(deck)
52
```

Reading specific cards from the deck—say, the first or the last—should be as easy as `deck[0]` or `deck[-1]`, and this is what the `__getitem__` method provides:

```
>>> deck[0]
Card(rank='2', suit='spades')
>>> deck[-1]
Card(rank='A', suit='hearts')
```

Should we create a method to pick a random card? No need. Python already has a function to get a random item from a sequence: `random.choice`. We can just use it on a deck instance:

```
>>> from random import choice
>>> choice(deck)
Card(rank='3', suit='hearts')
>>> choice(deck)
Card(rank='K', suit='spades')
>>> choice(deck)
Card(rank='2', suit='clubs')
```

We’ve just seen two advantages of using special methods to leverage the Python Data Model:

- Users of your classes don’t have to memorize arbitrary method names for standard operations (“How to get the number of items? Is it `.size()`, `.length()`, or what?”).
- It’s easier to benefit from the rich Python standard library and avoid reinventing the wheel, like the `random.choice` function.



But it gets better.

Because our `__getitem__` delegates to the `[]` operator of `self._cards`, our deck automatically supports slicing. Here's how we look at the top three cards from a brand new deck, and then pick just the Aces by starting at index 12 and skipping 13 cards at a time:

```
>>> deck[:3]
[Card(rank='2', suit='spades'), Card(rank='3',
suit='spades'),
Card(rank='4', suit='spades')]
>>> deck[12::13]
[Card(rank='A', suit='spades'), Card(rank='A',
suit='diamonds'),
Card(rank='A', suit='clubs'), Card(rank='A', suit='hearts')]
```

Just by implementing the `__getitem__` special method, our deck is also iterable:

```
>>> for card in deck: # doctest: +ELLIPSIS
...     print(card)
Card(rank='2', suit='spades')
Card(rank='3', suit='spades')
Card(rank='4', suit='spades')
...
```

The deck can also be iterated in reverse:

```
>>> for card in reversed(deck): # doctest: +ELLIPSIS
...     print(card)
Card(rank='A', suit='hearts')
Card(rank='K', suit='hearts')
Card(rank='Q', suit='hearts')
...
```

## ELLIPSIS IN DOCTESTS

Whenever possible, the Python console listings in this book were extracted from doctests to ensure accuracy. When the output was too long, the elided part is marked by an ellipsis (...) like in the last line in the preceding code. In such cases, we used the `# doctest: +ELLIPSIS` directive to make the doctest pass. If you are trying these examples in the interactive console, you may omit the doctest directives altogether.

Iteration is often implicit. If a collection has no `__contains__` method, the `in` operator does a sequential scan. Case in point: `in` works with our `FrenchDeck` class because it is iterable. Check it out:

```
>>> Card('Q', 'hearts') in deck
True
>>> Card('7', 'beasts') in deck
False
```

How about sorting? A common system of ranking cards is by rank (with aces being highest), then by suit in the order of spades (highest), then hearts, diamonds, and clubs (lowest). Here is a function that ranks cards by that rule, returning 0 for the 2 of clubs and 51 for the ace of spades:

```
suit_values = dict(spades=3, hearts=2, diamonds=1, clubs=0)

def spades_high(card):
    rank_value = FrenchDeck.ranks.index(card.rank)
    return rank_value * len(suit_values) +
    suit_values[card.suit]
```

Given `spades_high`, we can now list our deck in order of increasing rank:

```
>>> for card in sorted(deck, key=spades_high): # doctest:
+ELLIPSIS
...     print(card)
Card(rank='2', suit='clubs')
Card(rank='2', suit='diamonds')
Card(rank='2', suit='hearts')
... (46 cards omitted)
Card(rank='A', suit='diamonds')
Card(rank='A', suit='hearts')
Card(rank='A', suit='spades')
```

Although FrenchDeck implicitly inherits from `object`, its functionality is not inherited, but comes from leveraging the data model and composition. By implementing the special methods `__len__` and `__getitem__`, our FrenchDeck behaves like a standard Python sequence, allowing it to benefit from core language features (e.g., iteration and slicing) and from the standard library, as shown by the examples using `random.choice`, `reversed`, and `sorted`. Thanks to composition, the `__len__` and `__getitem__` implementations can delegate all the work to a list object, `self._cards`.

### HOW ABOUT SHUFFLING?

As implemented so far, a FrenchDeck cannot be shuffled, because it is *immutable*: the cards and their positions cannot be changed, except by violating encapsulation and handling the `_cards` attribute directly. In [\[Link to Come\]](#), we will fix that by adding a one-line `__setitem__` method.

## How Special Methods Are Used

The first thing to know about special methods is that they are meant to be called by the Python interpreter, and not by you. You don't write `my_object.__len__()`. You write `len(my_object)` and, if

`my_object` is an instance of a user-defined class, then Python calls the `__len__` method you implemented.

But the interpreter takes a shortcut when dealing for built-in types like `list`, `str`, `bytearray`, or extensions like the NumPy arrays. Python variable-sized collections written in C include a struct<sup>2</sup> called `PyVarObject`, which has an `ob_size` field holding the number of items in the collection. So, if `my_object` is an instance of one of those built-ins, then `len(my_object)` retrieves the value of the `ob_size` field, and this is much faster than calling a method.

More often than not, the special method call is implicit. For example, the statement `for i in x:` actually causes the invocation of `iter(x)`, which in turn may call `x.__iter__()` if that is available, or use `x.__getitem__()`--as in the `FrenchDeck` example.

Normally, your code should not have many direct calls to special methods. Unless you are doing a lot of metaprogramming, you should be implementing special methods more often than invoking them explicitly. The only special method that is frequently called by user code directly is `__init__`, to invoke the initializer of the superclass in your own `__init__` implementation.

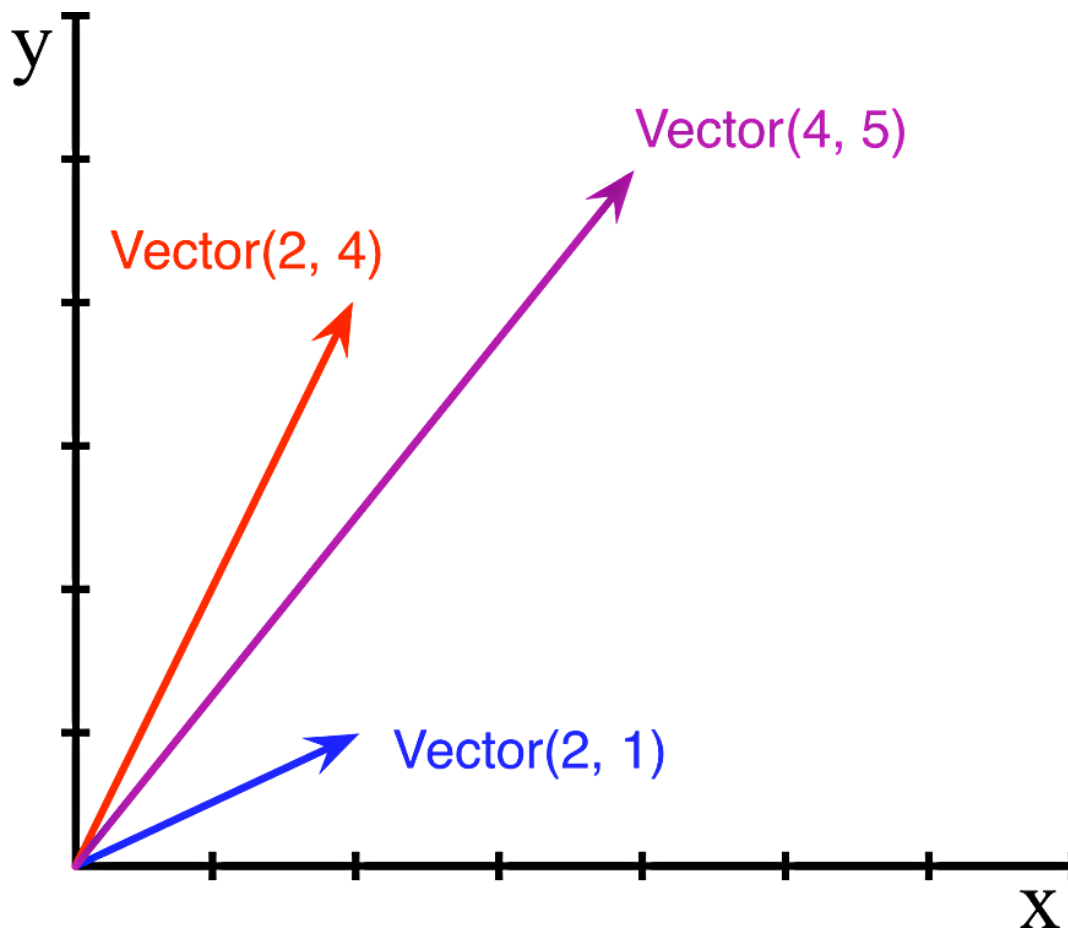
If you need to invoke a special method, it is usually better to call the related built-in function (e.g., `len`, `iter`, `str`, etc). These built-ins call the corresponding special method, but often provide other services and—for built-in types—are faster than method calls. See, for example, [Link to Come] in [Link to Come].

Avoid creating arbitrary, custom attributes with the `__foo__` syntax because such names may acquire special meanings in the future, even if they are unused today.

## Emulating Numeric Types

Several special methods allow user objects to respond to operators such as `+`. We will cover that in more detail in [\[Link to Come\]](#), but here our goal is to further illustrate the use of special methods through another simple example.

We will implement a class to represent two-dimensional vectors—that is Euclidean vectors like those used in math and physics (see [Figure 1-1](#)).



*Figure 1-1. Example of two-dimensional vector addition;  $\text{Vector}(2, 4) + \text{Vector}(2, 1)$  results in  $\text{Vector}(4, 5)$ .*

## TIP

The built-in `complex` type can be used to represent two-dimensional vectors, but our class can be extended to represent  $n$ -dimensional vectors. We will do that in [\[Link to Come\]](#).

We will start by designing the API for such a class by writing a simulated console session that we can use later as a doctest. The following snippet tests the vector addition pictured in [Figure 1-1](#):

```
>>> v1 = Vector(2, 4)
>>> v2 = Vector(2, 1)
>>> v1 + v2
Vector(4, 5)
```

Note how the `+` operator produces a `Vector` result, which is displayed in a friendly manner in the console.

The `abs` built-in function returns the absolute value of integers and floats, and the magnitude of `complex` numbers, so to be consistent, our API also uses `abs` to calculate the magnitude of a vector:

```
>>> v = Vector(3, 4)
>>> abs(v)
5.0
```

We can also implement the `*` operator to perform scalar multiplication (i.e., multiplying a vector by a number to produce a new vector with the same direction and a multiplied magnitude):

```
>>> v * 3
Vector(9, 12)
>>> abs(v * 3)
15.0
```

Example 1-2 is a `Vector` class implementing the operations just described, through the use of the special methods `__repr__`, `__abs__`, `__add__` and `__mul__`.

*Example 1-2. A simple two-dimensional vector class*

---

```
import math

class Vector:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __repr__(self):
        return f'Vector({self.x!r}, {self.y!r})'

    def __abs__(self):
        return math.hypot(self.x, self.y)

    def __bool__(self):
        return bool(abs(self))

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Vector(x, y)

    def __mul__(self, scalar):
        return Vector(self.x * scalar, self.y * scalar)
```

We implemented six special methods, but note that none of them is directly called within the class or in the typical usage of the class illustrated by the console listings. As mentioned before, the Python interpreter is the only frequent caller of most special methods. In the following sections, we discuss the code for the special methods in `Vector`.

## String Representation

The `__repr__` special method is called by the `repr` built-in to get the string representation of the object for inspection. If we did not implement `__repr__`, vector instances would be shown in the console like `<Vector object at 0x10e100070>`.

The interactive console and debugger call `repr` on the results of the expressions evaluated, as does the `%r` placeholder in classic formatting with the `%` operator, and the `!r` conversion field in the new Format String Syntax used in *f-strings* the `str.format` method.

Note that the *f-string* in our `__repr__`, uses `!r` to get the standard representation of the attributes to be displayed. This is good practice, because it shows the crucial difference between `Vector(1, 2)` and `Vector('1', '2')`—the latter would not work in the context of this example, because the constructor's arguments should be numbers, not `str`.

The string returned by `__repr__` should be unambiguous and, if possible, match the source code necessary to re-create the represented object. That is why our `Vector` representation looks like calling the constructor of the class (e.g., `Vector(3, 4)`).

Contrast `__repr__` with `__str__`, which is called by the `str()` constructor and implicitly used by the `print` function. `__str__` should return a string suitable for display to end users.

If you only implement one of these special methods, choose `__repr__`, because when no custom `__str__` is available, the `__str__` method inherited from the `object` class calls `__repr__` as a fallback.



### TIP

“Difference between `__str__` and `__repr__` in Python” is a Stack Overflow question with excellent contributions from Pythonistas Alex Martelli and Martijn Pieters.

## Arithmetic Operators

Example 1-2 implements two operators: `+` and `*`, to show basic usage of `__add__` and `__mul__`. Note that in both cases, the methods create and return a new instance of `Vector`, and do not modify either operand—`self` or `other` are merely read. This is the expected behavior of infix operators: to create new objects and not touch their operands. I will have a lot more to say about that in [Link to Come].

### WARNING

As implemented, Example 1-2 allows multiplying a `Vector` by a number, but not a number by a `Vector`, which violates the commutative property of scalar multiplication. We will fix that with the special method `__rmul__` in [Link to Come].

## Boolean Value of a Custom Type

Although Python has a `bool` type, it accepts any object in a boolean context, such as the expression controlling an `if` or `while` statement, or as operands to `and`, `or`, and `not`. To determine whether a value `x` is *truthy* or *falsy*, Python applies `bool(x)`, which always returns `True` or `False`.

By default, instances of user-defined classes are considered *truthy*, unless either `__bool__` or `__len__` is implemented. Basically,

`bool(x)` calls `x.__bool__()` and uses the result. If `__bool__` is not implemented, Python tries to invoke `x.__len__()`, and if that returns zero, `bool` returns `False`. Otherwise `bool` returns `True`.

Our implementation of `__bool__` is conceptually simple: it returns `False` if the magnitude of the vector is zero, `True` otherwise. We convert the magnitude to a Boolean using `bool(abs(self))` because `__bool__` is expected to return a boolean.

Note how the special method `__bool__` allows your objects to be consistent with the truth value testing rules defined in the “Built-in Types” chapter of *The Python Standard Library* documentation.

### NOTE

A faster implementation of `Vector.__bool__` is this:

```
def __bool__(self):  
    return bool(self.x or self.y)
```

This is harder to read, but avoids the trip through `abs`, `__abs__`, the squares, and square root. The explicit conversion to `bool` is needed because `__bool__` must return a boolean and `or` returns either operand as is: `x or y` evaluates to `x` if that is *truthy*, otherwise the result is `y`, whatever that is.

## Collection API

Figure 1-2 documents the interfaces of the essential collection types in the language. All the classes in the diagram are ABCs — *abstract base classes*. ABCs and the `collections.abc` module are covered in [Link to Come]. The goal of this brief section is to give a panoramic view of Python’s most important collection interfaces, showing how they are built from special methods.

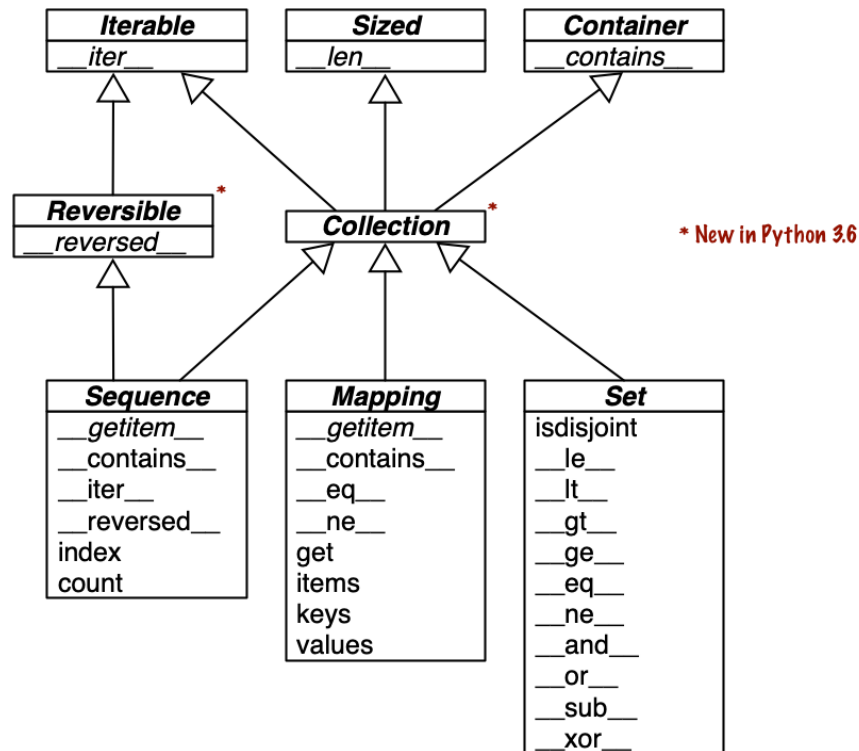


Figure 1-2. UML class diagram with fundamental collection types. Method names in *italic* are abstract, so they must be implemented by concrete subclasses such as *list* and *dict*. The remaining methods have concrete implementations, therefore subclasses can inherit them.

Each of the top ABCs has a single special method. The **Collection** ABC (new in Python 3.6) unifies the three essential interfaces that every collection should implement:

- **Iterable** to support `for`, comprehensions and other forms of iteration;
- **Sized** to support the `len` built-in function;
- **Contains** to support the `in` operator.

Python does not require concrete classes to actually inherit from any of these ABCs. Any class that implements `__len__` satisfies the **Sized** interface.

Three very important specializations of **Collection** are:

- **Sequence**, formalizing the interface of built-ins like `list`, and `str`;
- **Mapping**, implemented by `dict` and `collections.defaultdict`;
- **Set**: the interface of the `set` and `frozenset` built-ins.

Only **Sequence** is **Reversible**, because sequences support arbitrary ordering of their contents, while mappings and sets do not.

### NOTE

Since Python 3.7, the `dict` type is officially “ordered”, but that only means that the key insertion order is preserved. You cannot rearrange the keys in a `dict` however you like.

All but one of the special methods in the **Set ABC** implement infix operators. For example, `a & b` computes the intersection of sets `a` and `b`, and is implemented in the `__and__` special method.

The next three chapters will cover built-in sequences, mappings, and sets in detail.

Now let’s consider the major categories of special methods defined in the Python Data Model.

## Overview of Special Methods

The “Data Model” chapter of *The Python Language Reference* lists more than 80 special method names. More than half of them implement arithmetic, bitwise, and comparison operators. As an overview of what is available, see following tables.

Table 1-1 shows special method names excluding those used to implement infix operators or core math functions like `abs`. Most of these methods will be covered in detail throughout the book, including the most recent additions: asynchronous special methods such as `__anext__` (added in Python 3.5), and the class customization hook, `__init_subclass__` (from Python 3.6).

*Table 1-1. Special method names (operators excluded)*

Category	Method names
String/bytes representation	<code>__repr__</code> , <code>__str__</code> , <code>__format__</code> , <code>__bytes__</code> , <code>__fspath__</code>
Conversion to number	<code>__abs__</code> , <code>__bool__</code> , <code>__complex__</code> , <code>__int__</code> , <code>__float__</code> , <code>__hash__</code> , <code>__index__</code>
Emulating collections	<code>__len__</code> , <code>__getitem__</code> , <code>__setitem__</code> , <code>__delitem__</code> , <code>__contains__</code>
Iteration	<code>__iter__</code> , <code>__aiter__</code> , <code>__next__</code> , <code>__anext__</code> , <code>__reversed__</code>
Callable or coroutine execution	<code>__call__</code> , <code>__await__</code>
Context management	<code>__enter__</code> , <code>__aenter__</code> , <code>__exit__</code> , <code>__aexit__</code>
Instance creation and destruction	<code>__new__</code> , <code>__init__</code> , <code>__del__</code>
Attribute management	<code>__getattr__</code> , <code>__getattribute__</code> , <code>__setattr__</code> , <code>__delattr__</code> , <code>__dir__</code>
Attribute descriptors	<code>__get__</code> , <code>__set__</code> , <code>__delete__</code> , <code>__set_name__</code>
Class services	<code>__prepare__</code> , <code>__init_subclass__</code> , <code>__instancecheck__</code> , <code>__subclasscheck__</code>

Infix and numerical operators are supported by the special methods in 1-2. Here the most recent names are `__matmul__`, `__rmatmul__`, and `__imatmul__`, added in Python 3.5 to support the use of `@` as an infix operator for matrix multiplication, as we'll see in [Link to Come].

*Table 1-2. Special method names for operators*

Category	Method names and related operators
Unary numeric operators	<code>__neg__</code> -, <code>__pos__</code> +, <code>__abs__</code> <code>abs()</code>
Rich comparison operators	<code>__lt__</code> <, <code>__le__</code> <=, <code>__eq__</code> ==, <code>__ne__</code> !=, <code>__gt__</code> >, <code>__ge__</code> >=
Arithmetic operators	<code>__add__</code> +, <code>__sub__</code> -, <code>__mul__</code> *, <code>__truediv__</code> /, <code>__floordiv__</code> //, <code>__mod__</code> %, <code>__divmod__</code> <code>divmod()</code> , <code>__pow__</code> ** or <code>pow()</code> , <code>__round__</code> <code>round()</code> , <code>__matmul__</code> @
Reversed arithmetic operators	<code>__radd__</code> , <code>__rsub__</code> , <code>__rmul__</code> , <code>__rtruediv__</code> , <code>__rfloordiv__</code> , <code>__rmod__</code> , <code>__rdivmod__</code> , <code>__rpow__</code> , <code>__rmatmul__</code>
Augmented assignment arithmetic operators	<code>__iadd__</code> , <code>__isub__</code> , <code>__imul__</code> , <code>__itruediv__</code> , <code>__ifloordiv__</code> , <code>__imod__</code> , <code>__ipow__</code> , <code>__imatmul__</code>
Bitwise operators	<code>__invert__</code> ~, <code>__lshift__</code> <<, <code>__rshift__</code> >>, <code>__and__</code> &, <code>__or__</code>  , <code>__xor__</code> ^
Reversed bitwise operators	<code>__rlshift__</code> , <code>__rrshift__</code> , <code>__rand__</code> , <code>__rxor__</code> , <code>__ror__</code>
Augmented assignment bitwise operators	<code>__ilshift__</code> , <code>__irshift__</code> , <code>__iand__</code> , <code>__ixor__</code> , <code>__ior__</code>

## TIP

The reversed operators are fallbacks used when operands are swapped (`b * a` instead of `a * b`), while augmented assignments are shortcuts combining an infix operator with variable assignment (`a = a * b` becomes `a *= b`). [Link to Come] explains both reversed operators and augmented assignment in detail.

## Why len Is Not a Method

I asked this question to core developer Raymond Hettinger in 2013 and the key to his answer was a quote from The Zen of Python: “practicality beats purity.” In “How Special Methods Are Used”, I described how `len(x)` runs very fast when `x` is an instance of a built-in type. No method is called for the built-in objects of CPython: the length is simply read from a field in a C struct. Getting the number of items in a collection is a common operation and must work efficiently for such basic and diverse types as `str`, `list`, `memoryview`, and so on.

In other words, `len` is not called as a method because it gets special treatment as part of the Python Data Model, just like `abs`. But thanks to the special method `__len__`, you can also make `len` work with your own custom objects. This is a fair compromise between the need for efficient built-in objects and the consistency of the language. Also from The Zen of Python: “Special cases aren’t special enough to break the rules.”

## NOTE

If you think of `abs` and `len` as unary operators, you may be more inclined to forgive their functional look-and-feel, as opposed to the method call syntax one might expect in an OO language. In fact, the ABC language—a direct ancestor of Python that pioneered many of its features—had an `#` operator that was the equivalent of `len` (you'd write `#s`). When used as an infix operator, written `x#s`, it counted the occurrences of `x` in `s`, which in Python you get as `s.count(x)`, for any sequence `s`.



## Chapter Summary

By implementing special methods, your objects can behave like the built-in types, enabling the expressive coding style the community considers Pythonic.

A basic requirement for a Python object is to provide usable string representations of itself, one used for debugging and logging, another for presentation to end users. That is why the special methods `__repr__` and `__str__` exist in the data model.

Emulating sequences, as shown with the `FrenchDeck` example, is one of the most widely used applications of the special methods. Making the most of sequence types is the subject of [Chapter 2](#), and implementing your own sequence will be covered in [\[Link to Come\]](#) when we create a multidimensional extension of the `Vector` class.

Thanks to operator overloading, Python offers a rich selection of numeric types, from the built-ins to `decimal.Decimal` and `fractions.Fraction`, all supporting infix arithmetic operators. Implementing operators, including reversed operators and augmented assignment, will be shown in [\[Link to Come\]](#) via enhancements of the `Vector` example.

The use and implementation of the majority of the remaining special methods of the Python Data Model are covered throughout this book.

## Further Reading

The “Data Model” chapter of *The Python Language Reference* is the canonical source for the subject of this chapter and much of this book.

*Python in a Nutshell, 3rd Edition* (O’Reilly) by Alex Martelli, Anna Ravenscroft, and Steve Holden has excellent coverage of the data

model. Their description of the mechanics of attribute access is the most authoritative I've seen apart from the actual C source code of CPython. Martelli is also a prolific contributor to Stack Overflow, with more than 6,200 answers posted. See his user profile at [Stack Overflow](#).

David Beazley has two books covering the data model in detail in the context of Python 3: *Python Essential Reference, 4th Edition* (Addison-Wesley Professional), and *Python Cookbook, 3rd Edition* (O'Reilly), coauthored with Brian K. Jones.

*The Art of the Metaobject Protocol* (AMOP, MIT Press) by Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow explains the concept of a metaobject protocol, of which the Python Data Model is one example.

## SOAPBOX

### Data Model or Object Model?

What the Python documentation calls the “Python Data Model,” most authors would say is the “Python object model.” Martelli, Ravenscroft & Holden’s *Python in a Nutshell 3E*, and David Beazley’s *Python Essential Reference 4E* are the best books covering the “Python Data Model,” but they refer to it as the “object model.” On Wikipedia, the first definition of **object model** is “The properties of objects in general in a specific computer programming language.” This is what the “Python Data Model” is about. In this book, I will use “data model” because the documentation favors that term when referring to the Python object model, and because it is the title of the [chapter of \*The Python Language Reference\*](#) most relevant to our discussions.

### Magic Methods

The Ruby community calls their equivalent of the special methods *magic methods*. Many in the Python community adopt that term as well. I believe the special methods are actually the opposite of magic. Python and Ruby empower their users with a rich metaobject protocol that is not magic, enabling muggles like you and I to emulate many of the features available to core developers.

In contrast, consider Go. Some objects in that language have features that are magic, in the sense that we cannot emulate them in our own user-defined types. For example, Go arrays, strings, and maps support the use brackets for item access, as in `a[i]`. But there’s no way to make the `[]` notation work with a new collection type that you define. Even worse, Go has no user-level concept of an iterable interface or an iterator object, therefore its `for`/`range` syntax is limited to supporting five “magic” built-in types, including arrays, strings and maps.

Maybe in the future, the designers of Go will enhance its metaobject protocol. But currently, it is much more limited than what we have in Python or Ruby.

### Metaobjects

*The Art of the Metaobject Protocol (AMOP)* is my favorite computer book title. Less subjectively, the term *metaobject protocol* is useful to think about the Python Data Model and similar features in other languages. The *metaobject* part refers to the objects that are the building blocks of the language itself. In this context, *protocol* is a synonym of *interface*. So a *metaobject protocol* is a fancy synonym for object model: an API for core language constructs.

A rich metaobject protocol enables extending a language to support new programming paradigms. Gregor Kiczales, the first author of the *AMOP* book,

later became a pioneer in aspect-oriented programming and the initial author of AspectJ, an extension of Java implementing that paradigm. Aspect-oriented programming is much easier to implement in a dynamic language like Python, and several frameworks do it, but the most important is `zope.interface`, which is briefly discussed in [Link to Come] of [Link to Come].

- 
- 1 Story of Jython, written as a Foreword to *Jython Essentials* (O'Reilly, 2002), by Samuele Pedroni and Noel Rappin.
  - 2 A C struct is a record type with named fields.

## **Part II. Data Structures**

---

## Chapter 2. An Array of Sequences

---

### A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at [fluentpython2e@ramalho.org](mailto:fluentpython2e@ramalho.org).

*As you may have noticed, several of the operations mentioned work equally for texts, lists and tables. Texts, lists and tables together are called trains. [...] The FOR command also works generically on trains.*<sup>1</sup>

—Geurts, Meertens, and Pemberton, ABC  
Programmer’s Handbook

Before creating Python, Guido was a contributor to the ABC language—a 10-year research project to design a programming environment for beginners. ABC introduced many ideas we now consider “Pythonic”: generic operations on different types of sequences, built-in tuple and mapping types, structure by indentation, strong typing without variable declarations, and more. It’s no accident that Python is so user-friendly.

Python inherited from ABC the uniform handling of sequences. Strings, lists, byte sequences, arrays, XML elements, and database results share a rich set of common operations including iteration, slicing, sorting, and concatenation.

Understanding the variety of sequences available in Python saves us from reinventing the wheel, and their common interface inspires us to create APIs that properly support and leverage existing and future sequence types.

Most of the discussion in this chapter applies to sequences in general, from the familiar `list` to the `str` and `bytes` types that are new in Python 3. Specific topics on lists, tuples, arrays, and queues are also covered here, but the specifics of Unicode strings and byte sequences appear in [Chapter 4](#). Also, the idea here is to cover sequence types that are ready to use. Creating your own sequence types is the subject of [\[Link to Come\]](#).

## What's new in this chapter

The sequence types are a very stable part of Python, therefore most the changes here are not updates but improvements over the 1<sup>st</sup> edition of *Fluent Python*. The most significant are:

- New diagram and description of the internals of sequences, contrasting containers and flat sequences.
- Brief comparison of the performance and storage characteristics of `list` versus `tuple`.
- Caveats of tuples with mutable elements, and how to detect them if needed.
- Coverage of named tuples moved to [“Classic Named Tuples”](#) in [Chapter 5](#), where they are compared to the new data classes.

## Overview of Built-In Sequences

The standard library offers a rich selection of sequence types implemented in C:

## Container sequences

`list`, `tuple`, and `collections.deque` can hold items of different types, including nested containers.

## Flat sequences

`str`, `bytes`, `bytearray`, `memoryview`, and `array.array` hold items of one simple type.

A *container sequence* holds references to the objects it contains, which may be of any type, while a *flat sequence* stores the value of its contents in its own memory space, and not as distinct objects. See Figure 2-1.

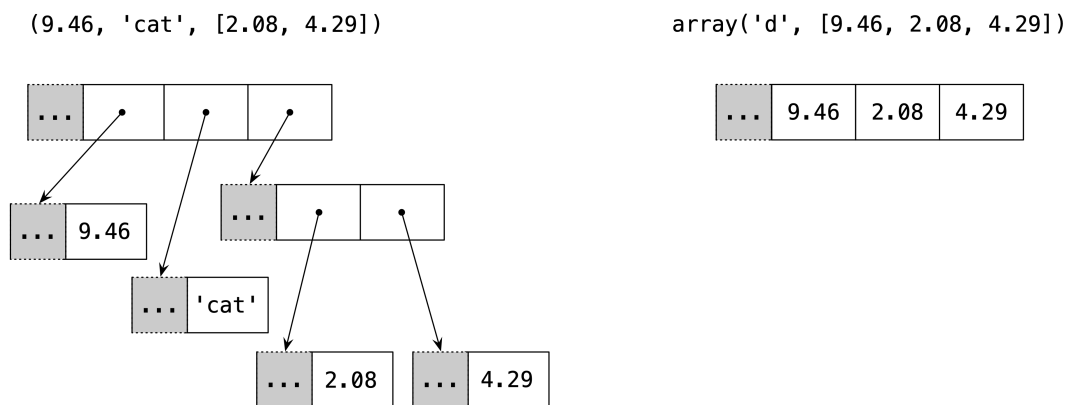


Figure 2-1. Simplified memory diagram of a *tuple* and an *array*, each holding 3 items. Gray cells represent the in-memory header of each Python object—not drawn to proportion. The *tuple* has an array of references to its contents. Each item is a separate Python object, possibly holding references to other Python objects, like that 2-item list. In contrast, the Python *array* is a single object, holding a C language array of 3 doubles.

Thus, flat sequences are more compact, but they are limited to holding primitive machine values like bytes, integers, and floats.



## NOTE

Every Python object in memory has a header with metadata. The simplest Python object, a `float`, has two metadata fields. On a 64-bit Python build, the struct representing a `float` has these 64-bit fields: \* `ob_refcnt`: the object's reference count; \* `ob_type`: a pointer to the object's type; \* `ob_fval`: a C `double` holding the value of the `float`. That's why an array with of floats is much more compact than a tuple of floats: the array is a single object holding the raw values of the floats, while the tuple is several objects—the tuple itself and each `float` object contained in it.

Another way of grouping sequence types is by mutability:

### *Mutable sequences*

`list`, `bytearray`, `array.array`, `collections.deque`, and `memoryview`

### *Immutable sequences*

`tuple`, `str`, and `bytes`

Figure 2-2 helps visualize how mutable sequences inherit all methods from immutable sequences, and implement several additional methods. The built-in concrete sequence types do not actually subclass the `Sequence` and `MutableSequence` abstract base classes (ABCs), but they are *virtual subclasses* registered with those ABCs—as we'll see in in [Link to Come]. Being virtual subclasses, `tuple` and `list` pass these tests:

```
>>> from collections import abc
>>> isinstance(tuple, abc.Sequence)
True
>>> isinstance(list, abc.MutableSequence)
True
```

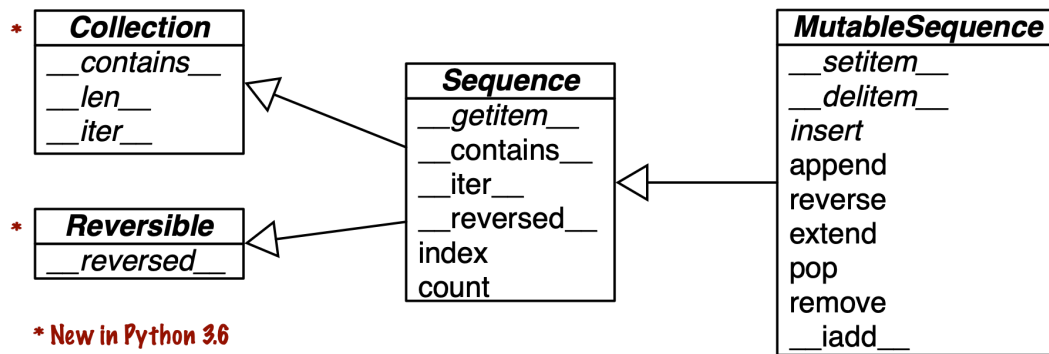


Figure 2-2. Simplified UML class diagram for some classes from `collections.abc` (superclasses are on the left; inheritance arrows point from subclasses to superclasses; names in italic are abstract classes and abstract methods)

Keep in mind these common traits: mutable versus immutable; container versus flat. They are helpful to extrapolate what you know about one sequence type to others.

The most fundamental sequence type is the `list`: a mutable container. I expect you are very familiar with them, so we'll jump right into list comprehensions, a powerful way of building lists that is sometimes underused because the syntax may look unusual at first. Mastering list comprehensions opens the door to generator expressions, which—among other uses—can produce elements to fill up sequences of any type. Both are the subject of the next section.

## List Comprehensions and Generator Expressions

A quick way to build a sequence is using a list comprehension (if the target is a `list`) or a generator expression (for all other kinds of sequences). If you are not using these syntactic forms on a daily basis, I bet you are missing opportunities to write code that is more readable and often faster at the same time.

If you doubt my claim that these constructs are “more readable,” read on. I'll try to convince you.

## TIP

For brevity, many Python programmers refer to list comprehensions as *listcomps*, and generator expressions as *genexps*. I will use these words as well.

## List Comprehensions and Readability

Here is a test: which do you find easier to read, [Example 2-1](#) or [Example 2-2](#)?

*Example 2-1. Build a list of Unicode codepoints from a string*

---

```
>>> symbols = '$ç£¥€α'
>>> codes = []
>>> for symbol in symbols:
...     codes.append(ord(symbol))
...
>>> codes
[36, 162, 163, 165, 8364, 164]
```

*Example 2-2. Build a list of Unicode codepoints from a string, take two*

---

```
>>> symbols = '$ç£¥€α'
>>> codes = [ord(symbol) for symbol in symbols]
>>> codes
[36, 162, 163, 165, 8364, 164]
```

Anybody who knows a little bit of Python can read [Example 2-1](#). However, after learning about listcomps, I find [Example 2-2](#) more readable because its intent is explicit.

A `for` loop may be used to do lots of different things: scanning a sequence to count or pick items, computing aggregates (sums, averages), or any number of other processing tasks. The code in

Example 2-1 is building up a list. In contrast, a listcomp is more explicit. Its goal is to build a new list.

Of course, it is possible to abuse list comprehensions to write truly incomprehensible code. I've seen Python code with listcomps used just to repeat a block of code for its side effects. If you are not doing something with the produced list, you should not use that syntax. Also, try to keep it short. If the list comprehension spans more than two lines, it is probably best to break it apart or rewrite as a plain old for loop. Use your best judgment: for Python as for English, there are no hard-and-fast rules for clear writing.

### SYNTAX TIP

In Python code, line breaks are ignored inside pairs of `[ ]`, `{ }`, or `( )`. So you can build multiline lists, listcomps, genexps, dictionaries and the like without using the ugly `\` line continuation escape. Also, when those delimiters are used to define a literal with a comma-separated series of items, a trailing comma will be ignored. So, for example, when coding a multi-line list literal, it is thoughtful to put a comma after the last item, making it a little easier for the next coder to add one more item to that list.

## LOCAL SCOPE WITHIN COMPREHENSIONS AND GENERATOR EXPRESSIONS

In Python 3, list comprehensions, generator expressions, and their siblings set and dict comprehensions have their own local scope, like functions. Variables assigned within the expression are local, but variables in the surrounding scope can still be referenced. Even better, the local variables do not mask the variables from the surrounding scope.

```
>>> x = 'ABC'
>>> codes = [ord(x) for x in x]
>>> x ❶
'ABC'
>>> codes ❷
[65, 66, 67]
>>>
```

- ❶ x is unchanged: it's still bound to 'ABC'.
- ❷ The list comprehension produces the expected list.

That code works, but I would not recommend using the same variable `x` to mean different things inside the listcomp.

List comprehensions build lists from sequences or any other iterable type by filtering and transforming items. The `filter` and `map` built-ins can be composed to do the same, but readability suffers, as we will see next.

## Listcomps Versus map and filter

Listcomps do everything the `map` and `filter` functions do, without the contortions of the functionally challenged Python `lambda`. Consider [Example 2-3](#).

*Example 2-3. The same list built by a listcomp and a map/filter composition*

```
>>> symbols = '$ç£¥€¤'
>>> beyond_ascii = [ord(s) for s in symbols if ord(s) > 127]
```

```
>>> beyond_ascii
[162, 163, 165, 8364, 164]
>>> beyond_ascii = list(filter(lambda c: c > 127, map(ord,
symbols)))
>>> beyond_ascii
[162, 163, 165, 8364, 164]
```

I used to believe that `map` and `filter` were faster than the equivalent listcomps, but Alex Martelli pointed out that's not the case—at least not in the preceding examples. The [\*02-array-seq/listcomp\\_speed.py\*](#) script in the *Fluent Python* code repository is a simple speed test comparing listcomp with `filter/map`.

I'll have more to say about `map` and `filter` in [Link to Come]. Now we turn to the use of listcomps to compute Cartesian products: a list containing tuples built from all items from two or more lists.

## Cartesian Products

Listcomps can build lists from the Cartesian product of two or more iterables. The items that make up the cartesian product are tuples made from items from every input iterable. The resulting list has a length equal to the lengths of the input iterables multiplied. See [Figure 2-3](#).

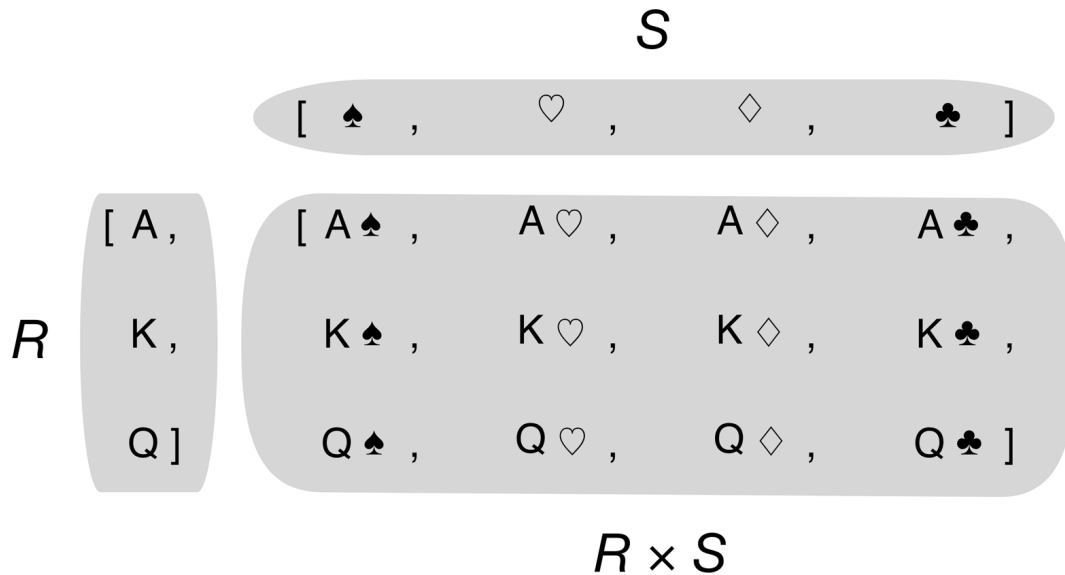


Figure 2-3. The Cartesian product of three card ranks and four suits is a sequence of twelve pairings

For example, imagine you need to produce a list of T-shirts available in two colors and three sizes. [Example 2-4](#) shows how to produce that list using a listcomp. The result has six items.

#### Example 2-4. Cartesian product using a list comprehension

```
>>> colors = ['black', 'white']
>>> sizes = ['S', 'M', 'L']
>>> tshirts = [(color, size) for color in colors for size in
sizes] ❶
>>> tshirts
[('black', 'S'), ('black', 'M'), ('black', 'L'), ('white', 'S'),
 ('white', 'M'), ('white', 'L')]
>>> for color in colors: ❷
...     for size in sizes:
...         print((color, size))
...
('black', 'S')
('black', 'M')
('black', 'L')
('white', 'S')
('white', 'M')
('white', 'L')
>>> tshirts = [(color, size) for size in sizes ❸
```

```
...                for color in colors]
>>> tshirts
[('black', 'S'), ('white', 'S'), ('black', 'M'), ('white', 'M'),
 ('black', 'L'), ('white', 'L')]
```

- ❶ This generates a list of tuples arranged by color, then size.
- ❷ Note how the resulting list is arranged as if the `for` loops were nested in the same order as they appear in the listcomp.
- ❸ To get items arranged by size, then color, just rearrange the `for` clauses; adding a line break to the listcomp makes it easy to see how the result will be ordered.

In [Example 1-1](#) (Chapter 1), the following expression was used to initialize a card deck with a list made of 52 cards from all 13 ranks of each of the 4 suits, sorted by suit then rank:

```
self._cards = [Card(rank, suit) for suit in
self.suits
                for rank in
self.ranks]
```

Listcomps are a one-trick pony: they build lists. To generate data for other sequence types, a genexp is the way to go. The next section is a brief look at genexps in the context of building nonlist sequences.

## Generator Expressions

To initialize tuples, arrays, and other types of sequences, you could also start from a listcomp, but a genexp saves memory because it yields items one by one using the iterator protocol instead of building a whole list just to feed another constructor.

Genexps use the same syntax as listcomps, but are enclosed in parentheses rather than brackets.



Example 2-5 shows basic usage of genexps to build a tuple and an array.

*Example 2-5. Initializing a tuple and an array from a generator expression*

---

```
>>> symbols = '$ç£¥€¤'
>>> tuple(ord(symbol) for symbol in symbols) ❶
(36, 162, 163, 165, 8364, 164)
>>> import array
>>> array.array('I', (ord(symbol) for symbol in symbols)) ❷
array('I', [36, 162, 163, 165, 8364, 164])
```

- ❶ If the generator expression is the single argument in a function call, there is no need to duplicate the enclosing parentheses.
- ❷ The `array` constructor takes two arguments, so the parentheses around the generator expression are mandatory. The first argument of the `array` constructor defines the storage type used for the numbers in the array, as we'll see in “Arrays”.

Example 2-6 uses a genexp with a Cartesian product to print out a roster of T-shirts of two colors in three sizes. In contrast with Example 2-4, here the six-item list of T-shirts is never built in memory: the generator expression feeds the `for` loop producing one item at a time. If the two lists used in the Cartesian product had 1,000 items each, using a generator expression would save the cost of building a list with a million items just to feed the `for` loop.

*Example 2-6. Cartesian product in a generator expression*

---

```
>>> colors = ['black', 'white']
>>> sizes = ['S', 'M', 'L']
>>> for tshirt in ('%s %s' % (c, s) for c in colors for s in
... sizes): ❶
...     print(tshirt)
...
black S
black M
black L
```

```
white S  
white M  
white L
```

- ❶ The generator expression yields items one by one; a list with all six T-shirt variations is never produced in this example.

[Link to Come] is devoted to explaining how generators work in detail. Here the idea was just to show the use of generator expressions to initialize sequences other than lists, or to produce output that you don't need to keep in memory.

Now we move on to the other fundamental sequence type in Python: the tuple.

## Tuples Are Not Just Immutable Lists

Some introductory texts about Python present tuples as “immutable lists,” but that is short selling them. Tuples do double duty: they can be used as immutable lists and also as records with no field names. This use is sometimes overlooked, so we will start with that.

### Tuples as Records

Tuples hold records: each item in the tuple holds the data for one field and the position of the item gives its meaning.

If you think of a tuple just as an immutable list, the quantity and the order of the items may or may not be important, depending on the context. But when using a tuple as a collection of fields, the number of items is usually fixed and their order is always important.

Example 2-7 shows tuples used as records. Note that in every expression, sorting the tuple would destroy the information because the meaning of each data item is given by its position in the tuple.

### Example 2-7. Tuples used as records

---

```
>>> lax_coordinates = (33.9425, -118.408056) ❶
>>> city, year, pop, chg, area = ('Tokyo', 2003, 32_450, 0.66,
8014) ❷
>>> traveler_ids = [('USA', '31195855'), ('BRA', 'CE342567'),
❸
...      ('ESP', 'XDA205856')]
>>> for passport in sorted(traveler_ids): ❹
...     print('%s/%s' % passport) ❺
...
BRA/CE342567
ESP/XDA205856
USA/31195855
>>> for country, _ in traveler_ids: ❻
...     print(country)
...
USA
BRA
ESP
```

- ❶ Latitude and longitude of the Los Angeles International Airport.
- ❷ Data about Tokyo: name, year, population (thousands), population change (%), area (km<sup>2</sup>).
- ❸ A list of tuples of the form (country\_code, passport\_number).
- ❹ As we iterate over the list, `passport` is bound to each tuple.
- ❺ The % formatting operator understands tuples and treats each item as a separate field.
- ❻ The `for` loop knows how to retrieve the items of a tuple separately—this is called “unpacking.” Here we are not interested in the second item, so it’s assigned to `_`, a dummy variable.

Tuples work well as records because of the unpacking mechanism—our next subject.

## Unpacking

In [Example 2-7](#), we assigned ('Tokyo', 2003, 32\_450, 0.66, 8014) to `city`, `year`, `pop`, `chg`, `area` in a single statement. Then, in the last line, the `%` operator assigned each item in the `passport` tuple to one slot in the format string in the `print` argument. Those are two examples of *tuple unpacking*.

### TIP

Tuple unpacking works with any iterable object. The only requirement is that the iterable yields exactly one item per variable in the receiving tuple, unless you use a star (\*) to capture excess items as explained in [“Using \\* to grab excess items”](#). The term *tuple unpacking* is widely used by Pythonistas, but *iterable unpacking* is gaining traction, as in the title of [PEP 3132 — Extended Iterable Unpacking](#).

The most visible form of unpacking is *parallel assignment*; that is, assigning items from an iterable to a tuple of variables, as you can see in this example:

```
>>> lax_coordinates = (33.9425, -118.408056)
>>> latitude, longitude = lax_coordinates # unpacking
>>> latitude
33.9425
>>> longitude
-118.408056
```

An elegant application of tuple unpacking is swapping the values of variables without using a temporary variable:

```
>>> b, a = a, b
```

Another example of unpacking is prefixing an argument with a star when calling a function:

```
>>> divmod(20, 8)
(2, 4)
>>> t = (20, 8)
>>> divmod(*t)
(2, 4)
>>> quotient, remainder = divmod(*t)
>>> quotient, remainder
(2, 4)
```

The preceding code also shows a further use of tuple unpacking: enabling functions to return multiple values in a way that is convenient to the caller. As another example, the `os.path.split()` function builds a tuple (`path`, `last_part`) from a filesystem path:

```
>>> import os
>>> _, filename =
os.path.split('/home/luciano/.ssh/idrsa.pub')
>>> filename
'idrsa.pub'
```

Sometimes when we only care about certain parts of a tuple when unpacking, a dummy variable like `_` is used as placeholder, as in the preceding example.

### WARNING

If you write internationalized software, `_` is not a good dummy variable because it is traditionally used as an alias to the `gettext.gettext` function, as recommended in the [gettext module documentation](#). Otherwise, it's a conventional name for a placeholder variable to be ignored.

Another way of using just some of the items when unpacking is to use the `*` syntax, as we'll see right away.

## USING `*` TO GRAB EXCESS ITEMS

Defining function parameters with `*args` to grab arbitrary excess arguments is a classic Python feature.

In Python 3, this idea was extended to apply to parallel assignment as well:

```
>>> a, b, *rest = range(5)
>>> a, b, rest
(0, 1, [2, 3, 4])
>>> a, b, *rest = range(3)
>>> a, b, rest
(0, 1, [2])
>>> a, b, *rest = range(2)
>>> a, b, rest
(0, 1, [])
```

In the context of parallel assignment, the `*` prefix can be applied to exactly one variable, but it can appear in any position:

```
>>> a, *body, c, d = range(5)
>>> a, body, c, d
(0, [1, 2], 3, 4)
>>> *head, b, c, d = range(5)
>>> head, b, c, d
([0, 1], 2, 3, 4)
```

Finally, a powerful feature of tuple unpacking is that it works with nested structures.

## Nested Tuple Unpacking

The tuple to receive an expression to unpack can have nested tuples, like `(a, b, (c, d))`, and Python will do the right thing if the expression matches the nesting structure. [Example 2-8](#) shows nested tuple unpacking in action.

*Example 2-8. Unpacking nested tuples to access the longitude*

---

```
metro_areas = [  
    ('Tokyo', 'JP', 36.933, (35.689722, 139.691667)), ❶  
    ('Delhi NCR', 'IN', 21.935, (28.613889, 77.208889)),  
    ('Mexico City', 'MX', 20.142, (19.433333, -99.133333)),  
    ('New York-Newark', 'US', 20.104, (40.808611, -74.020386)),  
    ('Sao Paulo', 'BR', 19.649, (-23.547778, -46.635833)),  
]  
  
print('{:15} | {:^9} | {:^9}'.format('', 'lat.', 'long.'))  
fmt = '{:15} | {:9.4f} | {:9.4f}'  
for name, cc, pop, (latitude, longitude) in metro_areas: ❷  
    if longitude <= 0: ❸  
        print(fmt.format(name, latitude, longitude))
```

- ❶ Each tuple holds a record with four fields, the last of which is a coordinate pair.
- ❷ By assigning the last field to a nested tuple, we unpack the coordinates.
- ❸ `if longitude <= 0:` limits the output to metropolitan areas in the Western hemisphere.

The output of [Example 2-8](#) is:

	lat.	long.
Mexico City	19.4333	-99.1333
New York-Newark	40.8086	-74.0204
Sao Paulo	-23.5478	-46.6358

## WARNING

Before Python 3, it was possible to define functions with nested tuples in the formal parameters (e.g., `def fn(a, (b, c), d):`). This is no longer supported in Python 3 function definitions, for practical reasons explained in [PEP 3113 — Removal of Tuple Parameter Unpacking](#). To be clear: nothing changed from the perspective of users calling a function. The restriction applies only to the definition of functions.

Python 3.5 introduced more flexible syntax for iterable unpacking, summarized in [What's New In Python 3.5](#).

As designed, tuples are very handy. But when using them as records, sometimes it is desirable to name the fields. The solution is the `namedtuple` factory we will cover in [Chapter 5](#).

Now let's consider the `tuple` class as an immutable variant of the `list` class.

## Tuples as Immutable Lists

The Python interpreter and standard library make extensive use of tuples as immutable lists, and so should you. This brings two key benefits:

- **Clarity:** when you see a `tuple` in code, you know its length will never change.
- **Performance:** a `tuple` uses less memory than a `list` of the same length, and they allow Python to do some optimizations.

However, be aware that the immutability of a `tuple` only applies to the references contained in it. References in a tuple cannot be deleted



or replaced. But if one of those references points to a mutable object, and that object is changed, then the value of the `tuple` changes. The next snippet demonstrates this point by creating two tuples—`a` and `b`—which are initially equal. When the last item in `b` is changed, they become different:

```
>>> a = (10, 'alpha', [1, 2])
>>> b = (10, 'alpha', [1, 2])
>>> a == b
True
>>> b[-1].append(99)
>>> a == b
False
>>> b
(10, 'alpha', [1, 2, 99])
```

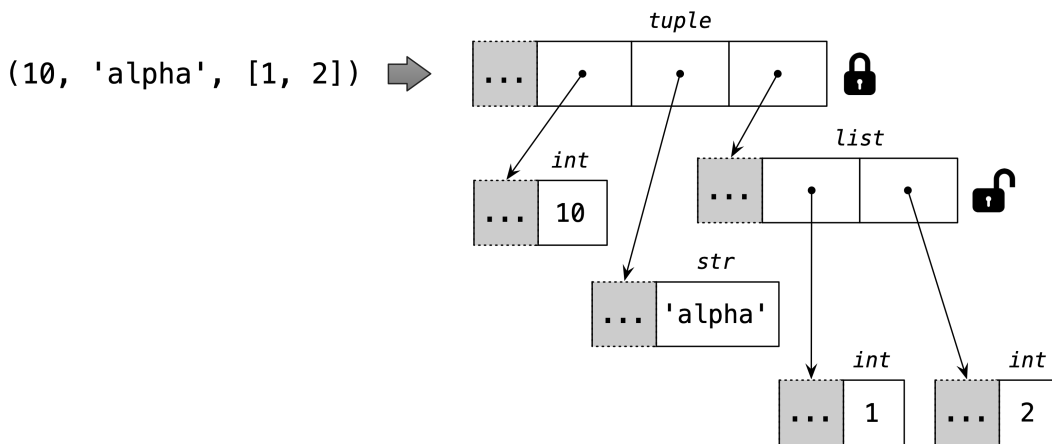


Figure 2-4. The content of the tuple itself is immutable, but that only means the references held by the tuple will always point to the same objects. However, if one of the referenced objects is a list, its content may change.

The mutable value of tuples with mutable items can be a source of bugs. If you want to make sure a `tuple` will stay unchanged, you can compute its hash. As we'll see in “What Is Hashable?”, an object is only hashable if its value cannot ever change. Therefore, here is a way to check that a tuple (or any object) has a fixed value:

```
>>> def fixed(o):
...     try:
...         hash(o)
...     except TypeError:
...         return False
...     return True
...
>>> tf = (10, 'alpha', (1, 2))
>>> tm = (10, 'alpha', [1, 2])
>>> fixed(tf)
True
>>> fixed(tm)
False
```

We explore this issue further in [“The Relative Immutability of Tuples”](#).

Despite this caveat, tuples are widely used as immutable lists. They offer some performance advantages explained by Python core developer Raymond Hettinger in a [StackOverflow answer](#) to the question [Are tuples more efficient than lists in Python?](#) To summarize, Hettinger wrote:

- To evaluate a tuple literal, the Python compiler generates bytecode for a tuple constant in one operation, but for a list literal, the generated bytecode pushes each element as a separate constant to the data stack, and then builds the list.
- Given a hashable tuple `t`, the `tuple(t)` constructor just returns a reference to the same `t`. There’s no need to copy, because if `t` is hashable, its value is fixed. In contrast, given a list `l`, the `list(l)` constructor must create a whole new copy of `l`.
- Because of its fixed length, a `tuple` instance is allocated the exact memory space it needs. Instances of `list`, on the

other hand, are allocated with room to spare, to amortize the cost of future appends.

- The references to the items in a tuple are stored in an array within the tuple struct itself, while a list holds a pointer to an array of references stored elsewhere. The added indirection makes CPU caches less effective, with potential impact on performance. But the indirection is necessary because when a list grows beyond the space currently allocated, the array of references needs to be relocated to make room.

## **Tuple versus list methods**

When using a tuple as an immutable variation of list, it is good to know how similar are their APIs. As you can see in [Table 2-1](#), `tuple` supports all `list` methods that do not involve adding or removing items, with one exception—tuple lacks the `__reversed__` method. However, that is just for optimization; `reversed(my_tuple)` works without it.

*Table 2-1. Methods and attributes found in list or tuple (methods implemented by object are omitted for brevity)*

	list	tuple	
<code>s.__add__(s2)</code>	•	•	<code>s + s2</code> —concatenation
<code>s.__iadd__(s2)</code>	•		<code>s += s2</code> —in-place concatenation
<code>s.append(e)</code>	•		Append one element after last
<code>s.clear()</code>	•		Delete all items
<code>s.__contains__(e)</code>	•	•	<code>e in s</code>
<code>s.copy()</code>	•		Shallow copy of the list
<code>s.count(e)</code>	•	•	Count occurrences of an element
<code>s.__delitem__(p)</code>	•		Remove item at position <code>p</code>
<code>s.extend(it)</code>	•		Append items from iterable <code>it</code>
<code>s.__getitem__(p)</code>	•	•	<code>s[p]</code> —get item at position
<code>s.__getnewargs__()</code>		•	Support for optimized serialization with <code>pickle</code>
<code>s.index(e)</code>	•	•	Find position of first occurrence of <code>e</code>
<code>s.insert(p, e)</code>	•		Insert element <code>e</code> before the item at position <code>p</code>
<code>s.__iter__()</code>	•	•	Get iterator
<code>s.__len__()</code>	•	•	<code>len(s)</code> —number of items
<code>s.__mul__(n)</code>	•	•	<code>s * n</code> —repeated concatenation
<code>s.__imul__(n)</code>	•		<code>s *= n</code> —in-place repeated concatenation
<code>s.__rmul__(n)</code>	•	•	<code>n * s</code> —reversed repeated concatenation <sup>a</sup>

	list	tuple
<code>s.pop([p])</code>	•	Remove and return last item or item at optional position <code>p</code>
<code>s.remove(e)</code>	•	Remove first occurrence of element <code>e</code> by value
<code>s.reverse()</code>	•	Reverse the order of the items in place
<code>s.__reversed__()</code>	•	Get iterator to scan items from last to first
<code>s.__setitem__(p, e)</code>	•	<code>s[p] = e</code> —put <code>e</code> in position <code>p</code> , overwriting existing item
<code>s.sort([key], [reverse])</code>	•	Sort items in place with optional keyword arguments <code>key</code> and <code>reverse</code>

[a](#) Reversed operators are explained in [Link to Come].

Every Python programmer knows that sequences can be sliced using the `s[a:b]` syntax. We now turn to some less well-known facts about slicing.

## Slicing

A common feature of `list`, `tuple`, `str`, and all sequence types in Python is the support of slicing operations, which are more powerful than most people realize.

In this section, we describe the *use* of these advanced forms of slicing. Their implementation in a user-defined class will be covered in [Link to Come], in keeping with our philosophy of covering ready-to-use classes in this part of the book, and creating new classes in [Link to Come].

## Why Slices and Range Exclude the Last Item

The Pythonic convention of excluding the last item in slices and ranges works well with the zero-based indexing used in Python, C, and many other languages. Some convenient features of the convention are:

- It's easy to see the length of a slice or range when only the stop position is given: `range(3)` and `my_list[:3]` both produce three items.
- It's easy to compute the length of a slice or range when start and stop are given: just subtract `stop - start`.
- It's easy to split a sequence in two parts at any index `x`, without overlapping: simply get `my_list[:x]` and `my_list[x:]`. For example:

```
>>> l = [10, 20, 30, 40, 50, 60]
>>> l[:2] # split at 2
[10, 20]
>>> l[2:]
[30, 40, 50, 60]
>>> l[:3] # split at 3
[10, 20, 30]
>>> l[3:]
[40, 50, 60]
```

But the best arguments for this convention were written by the Dutch computer scientist Edsger W. Dijkstra (see the last reference in [“Further Reading”](#)).

Now let's take a close look at how Python interprets slice notation.

## Slice Objects

This is no secret, but worth repeating just in case: `s[a:b:c]` can be used to specify a stride or step `c`, causing the resulting slice to skip items. The stride can also be negative, returning items in reverse. Three examples make this clear:

```
>>> s = 'bicycle'
>>> s[::3]
'bye'
>>> s[::-1]
'elcycib'
>>> s[::-2]
'eccb'
```

Another example was shown in [Chapter 1](#) when we used `deck[12::13]` to get all the aces in the unshuffled deck:

```
>>> deck[12::13]
[Card(rank='A', suit='spades'), Card(rank='A',
suit='diamonds'),
Card(rank='A', suit='clubs'), Card(rank='A', suit='hearts')]
```

The notation `a:b:c` is only valid within `[]` when used as the indexing or subscript operator, and it produces a slice object: `slice(a, b, c)`. As we will see in [\[Link to Come\]](#), to evaluate the expression `seq[start:stop:step]`, Python calls `seq.__getitem__(slice(start, stop, step))`. Even if you are not implementing your own sequence types, knowing about slice objects is useful because it lets you assign names to slices, just like spreadsheets allow naming of cell ranges.

Suppose you need to parse flat-file data like the invoice shown in [Example 2-9](#). Instead of filling your code with hardcoded slices, you can name them. See how readable this makes the `for` loop at the end of the example.

### Example 2-9. Line items from a flat-file invoice

---

```
>>> invoice = """
...
0.....6.....40.....52...55.....
.
... 1909  Pimoroni PiBrella                $17.50    3
$52.50
... 1489  6mm Tactile Switch x20           $4.95     2
$9.90
... 1510  Panavise Jr. - PV-201            $28.00    1
$28.00
... 1601  PiTFT Mini Kit 320x240          $34.95    1
$34.95
... """
>>> SKU = slice(0, 6)
>>> DESCRIPTION = slice(6, 40)
>>> UNIT_PRICE = slice(40, 52)
>>> QUANTITY = slice(52, 55)
>>> ITEM_TOTAL = slice(55, None)
>>> line_items = invoice.split('\n')[2:]
>>> for item in line_items:
...     print(item[UNIT_PRICE], item[DESCRIPTION])
...
$17.50    Pimoroni PiBrella
$4.95     6mm Tactile Switch x20
$28.00    Panavise Jr. - PV-201
$34.95    PiTFT Mini Kit 320x240
```

We'll come back to `slice` objects when we discuss creating your own collections in [Link to Come]. Meanwhile, from a user perspective, slicing includes additional features such as multidimensional slices and ellipsis (...) notation. Read on.

## Multidimensional Slicing and Ellipsis

The `[]` operator can also take multiple indexes or slices separated by commas. The `__getitem__` and `__setitem__` special methods that handle the `[]` operator simply receive the indices in `a[i, j]` as a



tuple. In other words, to evaluate `a[i, j]`, Python calls `a.__getitem__((i, j))`.

This is used, for instance, in the external NumPy package, where items of a two-dimensional `numpy.ndarray` can be fetched using the syntax `a[i, j]` and a two-dimensional slice obtained with an expression like `a[m:n, k:l]`. [Example 2-22](#) later in this chapter shows the use of this notation.

Except for `memoryview`, the built-in sequence types in Python are one-dimensional, so they support only one index or slice, and not a tuple of them.<sup>2</sup>

The ellipsis—written with three full stops (`...`) and not `...` (Unicode U+2026)—is recognized as a token by the Python parser. It is an alias to the `Ellipsis` object, the single instance of the `ellipsis` class.<sup>3</sup> As such, it can be passed as an argument to functions and as part of a slice specification, as in `f(a, ..., z)` or `a[i:...]`. NumPy uses `...` as a shortcut when slicing arrays of many dimensions; for example, if `x` is a four-dimensional array, `x[i, ...]` is a shortcut for `x[i, :, :, :]`. See the [Tentative NumPy Tutorial](#) to learn more about this.

At the time of this writing, I am unaware of uses of `Ellipsis` or multidimensional indexes and slices in the Python standard library. If you spot one, let me know. These syntactic features exist to support user-defined types and extensions such as NumPy.

Slices are not just useful to extract information from sequences; they can also be used to change mutable sequences in place—that is, without rebuilding them from scratch.

## Assigning to Slices

Mutable sequences can be grafted, excised, and otherwise modified in place using slice notation on the left side of an assignment statement or as the target of a `del` statement. The next few examples give an idea of the power of this notation:

```
>>> l = list(range(10))
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> l[2:5] = [20, 30]
>>> l
[0, 1, 20, 30, 5, 6, 7, 8, 9]
>>> del l[5:7]
>>> l
[0, 1, 20, 30, 5, 8, 9]
>>> l[3::2] = [11, 22]
>>> l
[0, 1, 20, 11, 5, 22, 9]
>>> l[2:5] = 100 ❶
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only assign an iterable
>>> l[2:5] = [100]
>>> l
[0, 1, 100, 22, 9]
```

- ❶ When the target of the assignment is a slice, the right side must be an iterable object, even if it has just one item.

Every coder knows that concatenation is a common operation with sequences. Introductory Python tutorials explain the use of `+` and `*` for that purpose, but there are some subtle details on how they work, which we cover next.

## Using `+` and `*` with Sequences

Python programmers expect that sequences support `+` and `*`. Usually both operands of `+` must be of the same sequence type, and neither of

them is modified but a new sequence of that same type is created as result of the concatenation.

To concatenate multiple copies of the same sequence, multiply it by an integer. Again, a new sequence is created:

```
>>> l = [1, 2, 3]
>>> l * 5
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> 5 * 'abcd'
'abcdabcdabcdabcdabcd'
```

Both + and \* always create a new object, and never change their operands.

### WARNING

Beware of expressions like `a * n` when `a` is a sequence containing mutable items because the result may surprise you. For example, trying to initialize a list of lists as `my_list = [[]] * 3` will result in a list with three references to the same inner list, which is probably not what you want.

The next section covers the pitfalls of trying to use \* to initialize a list of lists.

## Building Lists of Lists

Sometimes we need to initialize a list with a certain number of nested lists—for example, to distribute students in a list of teams or to represent squares on a game board. The best way of doing so is with a list comprehension, as in [Example 2-10](#).

*Example 2-10. A list with three lists of length 3 can represent a tic-tac-toe board*

```
>>> board = [['_'] * 3 for i in range(3)] ❶
>>> board
[['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]
>>> board[1][2] = 'X' ❷
>>> board
[['_', '_', '_'], ['_', '_', 'X'], ['_', '_', '_']]
```

- ❶ Create a list of three lists of three items each. Inspect the structure.
- ❷ Place a mark in row 1, column 2, and check the result.

A tempting but wrong shortcut is doing it like [Example 2-11](#).

*Example 2-11. A list with three references to the same list is useless*

```
>>> weird_board = [['_'] * 3] * 3 ❶
>>> weird_board
[['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]
>>> weird_board[1][2] = '0' ❷
>>> weird_board
[['_', '_', '0'], ['_', '_', '0'], ['_', '_', '0']]
```

- ❶ The outer list is made of three references to the same inner list. While it is unchanged, all seems right.
- ❷ Placing a mark in row 1, column 2, reveals that all rows are aliases referring to the same object.

The problem with [Example 2-11](#) is that, in essence, it behaves like this code:

```
row = ['_'] * 3
board = []
for i in range(3):
    board.append(row) ❶
```

- ❶ The same row is appended three times to board.

On the other hand, the list comprehension from [Example 2-10](#) is equivalent to this code:

```
>>> board = []
>>> for i in range(3):
...     row = ['_'] * 3 ❶
...     board.append(row)
...
>>> board
[['_', '_', '_'], ['_', '_', '_'], ['_', '_', '_']]
>>> board[2][0] = 'X'
>>> board ❷
[['_', '_', '_'], ['_', '_', '_'], ['X', '_', '_']]
```

- ❶ Each iteration builds a new row and appends it to board.
- ❷ Only row 2 is changed, as expected.

### TIP

If either the problem or the solution in this section are not clear to you, relax. [Chapter 6](#) was written to clarify the mechanics and pitfalls of references and mutable objects.

So far we have discussed the use of the plain + and \* operators with sequences, but there are also the += and \*= operators, which produce very different results depending on the mutability of the target sequence. The following section explains how that works.

## Augmented Assignment with Sequences

The augmented assignment operators += and \*= behave quite differently depending on the first operand. To simplify the discussion,

we will focus on augmented addition first (`+=`), but the concepts also apply to `*=` and to other augmented assignment operators.

The special method that makes `+=` work is `__iadd__` (for “in-place addition”). However, if `__iadd__` is not implemented, Python falls back to calling `__add__`. Consider this simple expression:

```
>>> a += b
```

If `a` implements `__iadd__`, that will be called. In the case of mutable sequences (e.g., `list`, `bytearray`, `array.array`), `a` will be changed in place (i.e., the effect will be similar to `a.extend(b)`). However, when `a` does not implement `__iadd__`, the expression `a += b` has the same effect as `a = a + b`: the expression `a + b` is evaluated first, producing a new object, which is then bound to `a`. In other words, the identity of the object bound to `a` may or may not change, depending on the availability of `__iadd__`.

In general, for mutable sequences, it is a good bet that `__iadd__` is implemented and that `+=` happens in place. For immutable sequences, clearly there is no way for that to happen.

What I just wrote about `+=` also applies to `*=`, which is implemented via `__imul__`. The `__iadd__` and `__imul__` special methods are discussed in [\[Link to Come\]](#).

Here is a demonstration of `*=` with a mutable sequence and then an immutable one:

```
>>> l = [1, 2, 3]
>>> id(l)
4311953800 ❶
>>> l *= 2
>>> l
[1, 2, 3, 1, 2, 3]
>>> id(l)
```

```

4311953800 ❷
>>> t = (1, 2, 3)
>>> id(t)
4312681568 ❸
>>> t *= 2
>>> id(t)
4301348296 ❹

```

- ❶ ID of the initial list
- ❷ After multiplication, the list is the same object, with new items appended
- ❸ ID of the initial tuple
- ❹ After multiplication, a new tuple was created

Repeated concatenation of immutable sequences is inefficient, because instead of just appending new items, the interpreter has to copy the whole target sequence to create a new one with the new items concatenated.<sup>4</sup>

We’ve seen common use cases for +=. The next section shows an intriguing corner case that highlights what “immutable” really means in the context of tuples.

## A += Assignment Puzzler

Try to answer without using the console: what is the result of evaluating the two expressions in Example 2-12?<sup>5</sup>

*Example 2-12. A riddle*

```

>>> t = (1, 2, [30, 40])
>>> t[2] += [50, 60]

```

What happens next? Choose the best answer:

- \*A.\* `t` becomes `(1, 2, [30, 40, 50, 60])`.
- \*B.\* `TypeError` is raised with the message `'tuple' object

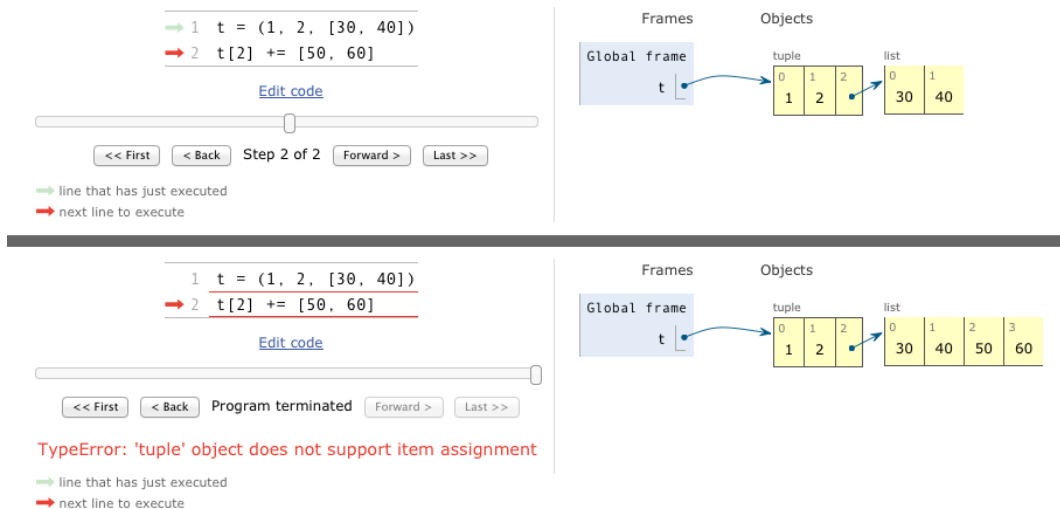
```
does not support item assignment`.
*C.* Neither.
*D.* Both *A* and *B*.
```

When I saw this, I was pretty sure the answer was **B**, but it's actually **D**, "Both **A** and **B**."! [Example 2-13](#) is the actual output from a Python 3.8 console.<sup>6</sup>

*Example 2-13. The unexpected result: item `t2` is changed and an exception is raised*

```
>>> t = (1, 2, [30, 40])
>>> t[2] += [50, 60]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> t
(1, 2, [30, 40, 50, 60])
```

[Online Python Tutor](#) is an awesome online tool to visualize how Python works in detail. [Figure 2-5](#) is a composite of two screenshots showing the initial and final states of the tuple `t` from [Example 2-13](#).



*Figure 2-5. Initial and final state of the tuple assignment puzzler (diagram generated by Online Python Tutor)*



If you look at the bytecode Python generates for the expression `s[a] += b` (Example 2-14), it becomes clear how that happens.

*Example 2-14. Bytecode for the expression `s[a] += b`*

```
>>> dis.dis('s[a] += b')
1          0 LOAD_NAME           0 (s)
          3 LOAD_NAME           1 (a)
          6 DUP_TOP_TWO
          7 BINARY_SUBSCR           ①
          8 LOAD_NAME           2 (b)
         11 INPLACE_ADD           ②
         12 ROT_THREE
         13 STORE_SUBSCR           ③
         14 LOAD_CONST          0 (None)
         17 RETURN_VALUE
```

- ① Put the value of `s[a]` on TOS (Top Of Stack).
- ② Perform `TOS += b`. This succeeds if TOS refers to a mutable object (it's a list, in Example 2-13).
- ③ Assign `s[a] = TOS`. This fails if `s` is immutable (the `t` tuple in Example 2-13).

This example is quite a corner case—in 20 years using Python, I have never seen this strange behavior actually bite somebody.

I take three lessons from this:

- Avoid putting mutable items in tuples.
- Augmented assignment is not an atomic operation—we just saw it throwing an exception after doing part of its job.
- Inspecting Python bytecode is not too difficult, and can be helpful to see what is going on under the hood.

After witnessing the subtleties of using `+` and `*` for concatenation, we can change the subject to another essential operation with sequences: sorting.

## list.sort and the sorted Built-In Function

The `list.sort` method sorts a list in-place—that is, without making a copy. It returns `None` to remind us that it changes the receiver<sup>7</sup> and does not create a new list. This is an important Python API convention: functions or methods that change an object in-place should return `None` to make it clear to the caller that the receiver was changed, and no new object was created. Similar behavior can be seen, for example, in the `random.shuffle(s)` function, which shuffles the mutable sequence `s` in-place, and returns `None`.

### NOTE

The convention of returning `None` to signal in-place changes has a drawback: we cannot cascade calls to those methods. In contrast, methods that return new objects (e.g., all `str` methods) can be cascaded in the fluent interface style. See Wikipedia’s [“Fluent interface”](#) entry for further description of this topic.

In contrast, the built-in function `sorted` creates a new list and returns it. In fact, it accepts any iterable object as an argument, including immutable sequences and generators (see [Link to Come]). Regardless of the type of iterable given to `sorted`, it always returns a newly created list.

Both `list.sort` and `sorted` take two optional, keyword-only arguments:

#### *reverse*

If `True`, the items are returned in descending order (i.e., by reversing the comparison of the items). The default is `False`.

#### *key*

A one-argument function that will be applied to each item to produce its sorting key. For example, when sorting a list of strings, `key=str.lower` can be used to perform a case-insensitive sort, and `key=len` will sort the strings by character length. The default is the identity function (i.e., the items themselves are compared).

### TIP

The `key` optional keyword parameter can also be used with the `min()` and `max()` built-ins and with other functions from the standard library (e.g., `itertools.groupby()` and `heapq.nlargest()`).

Here are a few examples to clarify the use of these functions and keyword arguments<sup>8</sup>:

```
>>> fruits = ['grape', 'raspberry', 'apple', 'banana']
>>> sorted(fruits)
['apple', 'banana', 'grape', 'raspberry'] ❶
>>> fruits
['grape', 'raspberry', 'apple', 'banana'] ❷
>>> sorted(fruits, reverse=True)
['raspberry', 'grape', 'banana', 'apple'] ❸
>>> sorted(fruits, key=len)
['grape', 'apple', 'banana', 'raspberry'] ❹
>>> sorted(fruits, key=len, reverse=True)
['raspberry', 'banana', 'grape', 'apple'] ❺
>>> fruits
['grape', 'raspberry', 'apple', 'banana'] ❻
>>> fruits.sort() ❼
>>> fruits
['apple', 'banana', 'grape', 'raspberry'] ❽
```

❶ This produces a new list of strings sorted alphabetically<sup>9</sup>.

- ❷ Inspecting the original list, we see it is unchanged.
- ❸ This is the previous “alphabetical” ordering, reversed.
- ❹ A new list of strings, now sorted by length. Because the sorting algorithm is stable, “grape” and “apple,” both of length 5, are in the original order.
- ❺ These are the strings sorted in descending order of length. It is not the reverse of the previous result because the sorting is stable, so again “grape” appears before “apple.”
- ❻ So far, the ordering of the original `fruits` list has not changed.
- ❼ This sorts the list in place, and returns `None` (which the console omits).
- ❽ Now `fruits` is sorted.

### WARNING

By default, Python sorts strings lexicographically by character code. That means ASCII uppercase letters will come before lowercase letters, and non-ASCII characters are unlikely to be sorted in a sensible way. “Sorting Unicode Text” covers proper ways of sorting text as humans would expect.

Once your sequences are sorted, they can be very efficiently searched. Fortunately, the standard binary search algorithm is already provided in the `bisect` module of the Python standard library. We discuss its essential features next, including the convenient `bisect.insort` function, which you can use to make sure that your sorted sequences stay sorted.

## Managing Ordered Sequences with `bisect`

The `bisect` module offers two main functions—`bisect` and `insort`—that use the binary search algorithm to quickly find and insert items in any sorted sequence.

## Searching with bisect

`bisect(haystack, needle)` does a binary search for `needle` in `haystack`—which must be a sorted sequence—to locate the position where `needle` can be inserted while maintaining `haystack` in ascending order. In other words, all items appearing up to that position are less than or equal to `needle`. You could use the result of `bisect(haystack, needle)` as the `index` argument to `haystack.insert(index, needle)`—however, using `insort` does both steps, and is faster.

### TIP

Raymond Hettinger wrote a `SortedCollection` recipe that leverages the `bisect` module and is easier to use than these standalone functions.

Example 2-15 uses a carefully chosen set of “needles” to demonstrate the insert positions returned by `bisect`. Its output is in Figure 2-6.

*Example 2-15. bisect finds insertion points for items in a sorted sequence*

---

```
import bisect
import sys

HAYSTACK = [1, 4, 5, 6, 8, 12, 15, 20, 21, 23, 23, 26, 29, 30]
NEEDLES = [0, 1, 2, 5, 8, 10, 22, 23, 29, 30, 31]

ROW_FMT = '{0:2d} @ {1:2d}      {2}{0:<2d}'

def demo(bisect_fn):
    for needle in reversed(NEEDLES):
        position = bisect_fn(HAYSTACK, needle) ❶
        offset = position * ' | ' ❷
        print(ROW_FMT.format(needle, position, offset)) ❸
```

```

if __name__ == '__main__':

    if sys.argv[-1] == 'left':    ❷
        bisect_fn = bisect.bisect_left
    else:
        bisect_fn = bisect

    print('DEMO:', bisect_fn.__name__)  ❸
    print('haystack ->', ' '.join('%2d' % n for n in HAYSTACK))
    demo(bisect_fn)

```

- ❶ Use the chosen `bisect` function to get the insertion point.
- ❷ Build a pattern of vertical bars proportional to the `offset`.
- ❸ Print formatted row showing needle and insertion point.
- ❹ Choose the `bisect` function to use according to the last command-line argument.
- ❺ Print header with name of function selected.

```

02-array-seq/ $ python3 bisect_demo.py
DEMO: bisect
haystack ->  1  4  5  6  8 12 15 20 21 23 23 26 29 30
31 @ 14      |  |  |  |  |  |  |  |  |  |  |  | 31
30 @ 14      |  |  |  |  |  |  |  |  |  |  |  | 30
29 @ 13      |  |  |  |  |  |  |  |  |  |  |  |29
23 @ 11      |  |  |  |  |  |  |  |  |  |  |23
22 @ 9       |  |  |  |  |  |  |  |  |  |22
10 @ 5       |  |  |  |  |10
 8 @ 5       |  |  |  | 8
 5 @ 3       |  | 15
 2 @ 1       |2
 1 @ 1       |1
 0 @ 0       0

```

Figure 2-6. Output of *Example 2-15* with `bisect` in use—each row starts with the notation `needle @ position` and the needle value appears again below its insertion point in the haystack

The behavior of `bisect` can be fine-tuned in two ways.

First, a pair of optional arguments, `lo` and `hi`, allow narrowing the region in the sequence to be searched when inserting. `lo` defaults to 0 and `hi` to the `len()` of the sequence.

Second, `bisect` is actually an alias for `bisect_right`, and there is a sister function called `bisect_left`. Their difference is apparent only when the needle compares equal to an item in the list: `bisect_right` returns an insertion point after the existing item, and `bisect_left` returns the position of the existing item, so insertion would occur before it. With simple types like `int`, inserting before or after makes no difference, but if the sequence contains objects that are distinct yet compare equal, then it may be relevant. For example, `1` and `1.0` are distinct, but `1 == 1.0` is `True`. Figure 2-7 shows the result of using `bisect_left`.

```
02-array-seq/ $ python3 bisect_demo.py left
DEMO: bisect_left
haystack -> 1 4 5 6 8 12 15 20 21 23 23 26 29 30
31 @ 14      | | | | | | | | | | | | | |31
30 @ 13      | | | | | | | | | | | | |30
29 @ 12      | | | | | | | | | | | |29
23 @ 9       | | | | | | | | |23
22 @ 9       | | | | | | | | |22
10 @ 5       | | | | |10
8 @ 4        | | |8
5 @ 2        |5
2 @ 1        |2
1 @ 0        1
0 @ 0        0
```

Figure 2-7. Output of Example 2-15 with `bisect_left` in use (compare with Figure 2-6 and note the insertion points for the values 1, 8, 23, 29, and 30 to the left of the same numbers in the haystack).

An interesting application of `bisect` is to perform table lookups by numeric values—for example, to convert test scores to letter grades, as in Example 2-16.

*Example 2-16. Given a test score, grade returns the corresponding letter grade*

---

```
>>> def grade(score, breakpoints=[60, 70, 80, 90],
grades='FDCBA'):
...     i = bisect.bisect(breakpoints, score)
...     return grades[i]
...
>>> [grade(score) for score in [55, 60, 65, 70, 75, 80, 85, 90,
95]]
['F', 'D', 'D', 'C', 'C', 'B', 'B', 'A', 'A']
```

The code in [Example 2-16](#) is from the [bisect module documentation](#), which also lists functions to use `bisect` as a faster replacement for the `index` method when searching through long ordered sequences of numbers.

When used for table lookups, `bisect_left` produces very different results<sup>10</sup>. Note the letter grade results in [Example 2-17](#).

*Example 2-17. `bisect_left` maps a score of 60 to grade F, not D as in [Example 2-16](#).*

---

```
>>> def grade(score, breakpoints=[60, 70, 80, 90],
grades='FDCBA'):
...     i = bisect.bisect_left(breakpoints, score)
...     return grades[i]
...
>>> [grade(score) for score in [55, 60, 65, 70, 75, 80, 85, 90,
95]]
['F', 'F', 'D', 'D', 'C', 'C', 'B', 'B', 'A']
```

These functions are not only used for searching, but also for inserting items in sorted sequences, as the following section shows.

## Inserting with `bisect.insort`



Sorting is expensive, so once you have a sorted sequence, it's good to keep it that way. That is why `bisect.insort` was created.

`insert(seq, item)` inserts `item` into `seq` so as to keep `seq` in ascending order. See [Example 2-18](#) and its output in [Figure 2-8](#).

*Example 2-18. Insert keeps a sorted sequence always sorted*

---

```
import bisect
import random

SIZE = 7

random.seed(1729)

my_list = []
for i in range(SIZE):
    new_item = random.randrange(SIZE*2)
    bisect.insort(my_list, new_item)
    print('%2d ->' % new_item, my_list)
```

```
02-array-seq/ $ python3 bisect_insert.py
10 -> [10]
0 -> [0, 10]
6 -> [0, 6, 10]
8 -> [0, 6, 8, 10]
7 -> [0, 6, 7, 8, 10]
2 -> [0, 2, 6, 7, 8, 10]
10 -> [0, 2, 6, 7, 8, 10, 10]
```

*Figure 2-8. Output of [Example 2-18](#)*

Like `bisect`, `insert` takes optional `lo`, `hi` arguments to limit the search to a sub-sequence. There is also an `insert_left` variation that uses `bisect_left` to find insertion points.

Much of what we have seen so far in this chapter applies to sequences in general, not just lists or tuples. Python programmers sometimes overuse the `list` type because it is so handy—I know I’ve done it. If you are handling lists of numbers, arrays are the way to go. The remainder of the chapter is devoted alternatives to lists and tuples.

## When a List Is Not the Answer

The `list` type is flexible and easy to use, but depending on specific requirements, there are better options. For example, an `array` saves a lot of memory when you need to store millions of floating-point values. On the other hand, if you are constantly adding and removing items from opposite ends of a list, it’s good to know that a `deque` (double-ended queue) is a more efficient FIFO data structure.

### TIP

If your code frequently checks whether an item is present in a collection (e.g., `item in my_collection`), consider using a `set` for `my_collection`, especially if it holds a large number of items. Sets are optimized for fast membership checking. But they are not sequences (their content is unordered). We cover them in [Chapter 3](#).

For the remainder of this chapter, we discuss mutable sequence types that can replace lists in many cases, starting with arrays.

## Arrays

If a list will only contain numbers, an `array.array` is more efficient: it supports all mutable sequence operations (including `.pop`, `.insert`, and `.extend`), and additional methods for fast loading and saving such as `.frombytes` and `.tofile`.

A Python array is as lean as a C array. As shown in [Figure 2-1](#), an array of `float` values does not hold full-fledged `float` instances, but only the packed bytes representing their machine values—similar to an array of `double` in the C language. When creating an array, you provide a typecode, a letter to determine the underlying C type used to store each item in the array. For example, `b` is the typecode for signed `char`. If you create an `array('b')`, then each item will be stored in a single byte and interpreted as an integer from `-128` to `127`. For large sequences of numbers, this saves a lot of memory. And Python will not let you put any number that does not match the type for the array.

[Example 2-19](#) shows creating, saving, and loading an array of 10 million floating-point random numbers.

*Example 2-19. Creating, saving, and loading a large array of floats*

---

```
>>> from array import array ❶
>>> from random import random
>>> floats = array('d', (random() for i in range(10**7))) ❷
>>> floats[-1] ❸
0.07802343889111107
>>> fp = open('floats.bin', 'wb')
>>> floats.tofile(fp) ❹
>>> fp.close()
>>> floats2 = array('d') ❺
>>> fp = open('floats.bin', 'rb')
>>> floats2.fromfile(fp, 10**7) ❻
>>> fp.close()
>>> floats2[-1] ❼
0.07802343889111107
>>> floats2 == floats ❽
True
```

- ❶ Import the `array` type.
- ❷ Create an array of double-precision floats (typecode `'d'`) from any iterable object—in this case, a generator expression.
- ❸ Inspect the last number in the array.

- ④ Save the array to a binary file.
- ⑤ Create an empty array of doubles.
- ⑥ Read 10 million numbers from the binary file.
- ⑦ Inspect the last number in the array.
- ⑧ Verify that the contents of the arrays match.

As you can see, `array.tofile` and `array.fromfile` are easy to use. If you try the example, you'll notice they are also very fast. A quick experiment shows that it takes about 0.1s for `array.fromfile` to load 10 million double-precision floats from a binary file created with `array.tofile`. That is nearly 60 times faster than reading the numbers from a text file, which also involves parsing each line with the `float` built-in. Saving with `array.tofile` is about 7 times faster than writing one float per line in a text file. In addition, the size of the binary file with 10 million doubles is 80,000,000 bytes (8 bytes per double, zero overhead), while the text file has 181,515,739 bytes, for the same data.

### TIP

Another fast—but more flexible—way of saving numeric data is the `pickle` module for object serialization. Saving an array of floats with `pickle.dump` is almost as fast as with `array.tofile`. However, `pickle` automatically handles almost all built-in types, including nested containers, and even instances of user-defined classes (if they are not too tricky in their implementation).

For the specific case of numeric arrays representing binary data, such as raster images, Python has the `bytes` and `bytearray` types discussed in [Chapter 4](#).

We wrap up this section on arrays with [Table 2-2](#), comparing the features of `list` and `array.array`.

*Table 2-2. Methods and attributes found in list or array (deprecated array methods and those also implemented by object were omitted for brevity)*

	list	array	
<code>s.__add__(s2)</code>	•	•	<code>s + s2</code> —concatenation
<code>s.__iadd__(s2)</code>	•	•	<code>s += s2</code> —in-place concatenation
<code>s.append(e)</code>	•	•	Append one element after last
<code>s.byteswap()</code>		•	Swap bytes of all items in array for endianness conversion
<code>s.clear()</code>	•		Delete all items
<code>s.__contains__(e)</code>	•	•	<code>e in s</code>
<code>s.copy()</code>	•		Shallow copy of the list
<code>s.__copy__()</code>		•	Support for <code>copy.copy</code>
<code>s.count(e)</code>	•	•	Count occurrences of an element
<code>s.__deepcopy__()</code>		•	Optimized support for <code>copy.deepcopy</code>
<code>s.__delitem__(p)</code>	•	•	Remove item at position <code>p</code>
<code>s.extend(it)</code>	•	•	Append items from iterable <code>it</code>
<code>s.frombytes(b)</code>		•	Append items from byte sequence interpreted as packed machine values
<code>s.fromfile(f, n)</code>		•	Append <code>n</code> items from binary file <code>f</code> interpreted as packed machine values
<code>s.fromlist(l)</code>		•	Append items from list; if one causes <code>TypeError</code> , none are appended
<code>s.__getitem__(p)</code>	•	•	<code>s[p]</code> —get item at position

	list	array	
<code>s.index(e)</code>	•	•	Find position of first occurrence of <code>e</code>
<code>s.insert(p, e)</code>	•	•	Insert element <code>e</code> before the item at position <code>p</code>
<code>s.itemsize</code>		•	Length in bytes of each array item
<code>s.__iter__()</code>	•	•	Get iterator
<code>s.__len__()</code>	•	•	<code>len(s)</code> —number of items
<code>s.__mul__(n)</code>	•	•	<code>s * n</code> —repeated concatenation
<code>s.__imul__(n)</code>	•	•	<code>s *= n</code> —in-place repeated concatenation
<code>s.__rmul__(n)</code>	•	•	<code>n * s</code> —reversed repeated concatenation <sup>a</sup>
<code>s.pop([p])</code>	•	•	Remove and return item at position <code>p</code> (default: last)
<code>s.remove(e)</code>	•	•	Remove first occurrence of element <code>e</code> by value
<code>s.reverse()</code>	•	•	Reverse the order of the items in place
<code>s.__reversed__()</code>	•		Get iterator to scan items from last to first
<code>s.__setitem__(p, e)</code>	•	•	<code>s[p] = e</code> —put <code>e</code> in position <code>p</code> , overwriting existing item
<code>s.sort([key], [reverse])</code>	•		Sort items in place with optional keyword arguments <code>key</code> and <code>reverse</code>
<code>s.tobytes()</code>		•	Return items as packed machine values in a bytes object
<code>s.tofile(f)</code>		•	Save items as packed machine values to binary file <code>f</code>
<code>s.tolist()</code>		•	Return items as numeric objects in a list

	list	array
<code>s.typecode</code>	•	One-character string identifying the C type of the items

a Reversed operators are explained in [\[Link to Come\]](#).

### TIP

As of Python 3.8, the `array` type does not have an in-place `sort` method like `list.sort()`. If you need to sort an array, use the `sorted` function to rebuild it sorted:

```
a = array.array(a.typecode, sorted(a))
```

To keep a sorted array sorted while adding items to it, use the `bisect.insort` function (as seen in [“Inserting with bisect.insort”](#)).

If you do a lot of work with arrays and don’t know about `memoryview`, you’re missing out. See the next topic.

## Memory Views

The built-in `memoryview` class is a shared-memory sequence type that lets you handle slices of arrays without copying bytes. It was inspired by the NumPy library (which we’ll discuss shortly in [“NumPy and SciPy”](#)). Travis Oliphant, lead author of NumPy, answers [When should a memoryview be used?](#) like this:

*A memoryview is essentially a generalized NumPy array structure in Python itself (without the math). It allows you to share memory between data-structures (things like PIL images, SQLite databases, NumPy arrays, etc.) without first copying. This is very important for large data sets.*

Using notation similar to the `array` module, the `memoryview.cast` method lets you change the way multiple bytes are read or written as units without moving bits around. `memoryview.cast` returns yet another `memoryview` object, always sharing the same memory.

Example 2-20 shows how to create alternate views on the same array of 6 bytes, to operate on it as  $2 \times 3$  matrix or a  $3 \times 2$  matrix:

*Example 2-20. Handling 6 bytes memory of as  $1 \times 6$ ,  $2 \times 3$ , and  $3 \times 2$  views*

```
>>> from array import array
>>> octets = array('B', range(6)) ❶
>>> m1 = memoryview(octets) ❷
>>> m1.tolist()
[0, 1, 2, 3, 4, 5]
>>> m2 = m1.cast('B', [2, 3]) ❸
>>> m2.tolist()
[[0, 1, 2], [3, 4, 5]]
>>> m3 = m1.cast('B', [3, 2]) ❹
>>> m3.tolist()
[[0, 1], [2, 3], [4, 5]]
>>> m2[1,1] = 22 ❺
>>> m3[1,1] = 33 ❻
>>> octets ❼
array('B', [0, 1, 2, 33, 22, 5])
```

- ❶ Build array of 6 bytes (typecode 'B').
- ❷ Build `memoryview` from that array, then export it as list.
- ❸ Build new `memoryview` from that previous one, but with 2 rows and 3 columns.
- ❹ Yet another `memoryview`, now with 3 rows and 2 columns.



- ⑤ Overwrite byte in `m2` at row 1, column 1 with 22.
- ⑥ Overwrite byte in `m3` at row 1, column 1 with 33.
- ⑦ Display original array, proving that the memory was shared among `octets`, `m1`, `m2`, and `m3`.

Indexing a `memoryview` using a tuple—as in the expression `m2[1,1]` above—is a feature that was added in Python 3.5.

The awesome power of `memoryview` can also be used to corrupt. [Example 2-21](#) shows how to change a single byte of an item in an array of 16-bit integers.

*Example 2-21. Changing the value of an 16-bit integer array item by poking one of its bytes*

```
>>> numbers = array.array('h', [-2, -1, 0, 1, 2])
>>> memv = memoryview(numbers) ❶
>>> len(memv)
5
>>> memv[0] ❷
-2
>>> memv_oct = memv.cast('B') ❸
>>> memv_oct.tolist() ❹
[254, 255, 255, 255, 0, 0, 1, 0, 2, 0]
>>> memv_oct[5] = 4 ❺
>>> numbers
array('h', [-2, -1, 1024, 1, 2]) ❻
```

- ❶ Build `memoryview` from array of 5 16-bit signed integers (typecode 'h').
- ❷ `memv` sees the same 5 items in the array.
- ❸ Create `memv_oct` by casting the elements of `memv` to bytes (typecode 'B').
- ❹ Export elements of `memv_oct` as a list of 10 bytes, for inspection.
- ❺ Assign value 4 to byte offset 5.
- ❻ Note change to `numbers`: a 4 in the most significant byte of a 2-byte unsigned integer is 1024.

We'll see another short example with `memoryview` in the context of binary sequence manipulations with `struct` ([Chapter 4, Example 5-25](#)).

Meanwhile, if you are doing advanced numeric processing in arrays, you should be using the NumPy and SciPy libraries. We'll take a brief look at them right away.

## NumPy and SciPy

Throughout this book, I make a point of highlighting what is already in the Python standard library so you can make the most of it. But NumPy and SciPy are so awesome that a detour is warranted.

For advanced array and matrix operations, NumPy and SciPy are the reason why Python became mainstream in scientific computing applications. NumPy implements multi-dimensional, homogeneous arrays and matrix types that hold not only numbers but also user-defined records, and provides efficient elementwise operations.

SciPy is a library, written on top of NumPy, offering many scientific computing algorithms from linear algebra, numerical calculus, and statistics. SciPy is fast and reliable because it leverages the widely used C and Fortran code base from the [Netlib Repository](#). In other words, SciPy gives scientists the best of both worlds: an interactive prompt and high-level Python APIs, together with industrial-strength number-crunching functions optimized in C and Fortran.

As a very brief demo, [Example 2-22](#) shows some basic operations with two-dimensional arrays in NumPy.

*Example 2-22. Basic operations with rows and columns in a `numpy.ndarray`*

---

```
>>> import numpy ❶
>>> a = numpy.arange(12) ❷
```

```

>>> a
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> type(a)
<class 'numpy.ndarray'>
>>> a.shape ❸
(12,)
>>> a.shape = 3, 4 ❹
>>> a
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> a[2] ❺
array([ 8,  9, 10, 11])
>>> a[2, 1] ❻
9
>>> a[:, 1] ❼
array([1, 5, 9]) ❽
>>> a.transpose()
array([[ 0,  4,  8],
       [ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11]])

```

- ❶ Import NumPy, after installing (it's not in the Python standard library).
- ❷ Build and inspect a `numpy.ndarray` with integers 0 to 11.
- ❸ Inspect the dimensions of the array: this is a one-dimensional, 12-element array.
- ❹ Change the shape of the array, adding one dimension, then inspecting the result.
- ❺ Get row at index 2.
- ❻ Get element at index 2, 1.
- ❼ Get column at index 1.
- ❽ Create a new array by transposing (swapping columns with rows).

NumPy also supports high-level operations for loading, saving, and operating on all elements of a `numpy.ndarray`:

```

>>> import numpy
>>> floats = numpy.loadtxt('floats-10M-lines.txt') ❶
>>> floats[-3:] ❷
array([ 3016362.69195522,   535281.10514262,
        4566560.44373946])
>>> floats *= .5 ❸
>>> floats[-3:]
array([ 1508181.34597761,   267640.55257131,
        2283280.22186973])
>>> from time import perf_counter as pc ❹
>>> t0 = pc(); floats /= 3; pc() - t0 ❺
0.03690556302899495
>>> numpy.save('floats-10M', floats) ❻
>>> floats2 = numpy.load('floats-10M.npy', 'r+') ❼
>>> floats2 *= 6
>>> floats2[-3:] ❽
memmap([ 3016362.69195522,   535281.10514262,
        4566560.44373946])

```

- ❶ Load 10 million floating-point numbers from a text file.
- ❷ Use sequence slicing notation to inspect the last three numbers.
- ❸ Multiply every element in the `floats` array by `.5` and inspect the last three elements again.
- ❹ Import the high-resolution performance measurement timer (available since Python 3.3).
- ❺ Divide every element by 3; the elapsed time for 10 million floats is less than 40 milliseconds.
- ❻ Save the array in a `.npy` binary file.
- ❼ Load the data as a memory-mapped file into another array; this allows efficient processing of slices of the array even if it does not fit entirely in memory.
- ❽ Inspect the last three elements after multiplying every element by 6.

## TIP

Installing NumPy and SciPy from source is may be challenging. The [Installing the SciPy Stack](#) page on SciPy.org recommends using special scientific Python distributions such as Anaconda, Enthought Canopy, and WinPython, among others. These are large downloads, but come ready to use. Users of popular GNU/Linux distributions can usually find NumPy and SciPy in the standard package repositories. For example, you can use this command to install them on Ubuntu:

```
$ sudo apt install python-numpy python-scipy
```

This was just an appetizer.

NumPy and SciPy are formidable libraries, and are the foundation of other awesome tools such as the [Pandas](#)—which implements efficient array types that can hold nonnumeric data and provides import/export functions for many different formats like *.csv*, *.xls*, SQL dumps, HDF5, etc.—and [Scikit-learn](#)—currently the most widely used Machine Learning toolset. Most NumPy and SciPy functions are implemented in C or C++, and can leverage all CPU cores because they release Python’s GIL (Global Interpreter Lock). The [Dask](#) project supports parallelizing NumPy, Pandas, and Scikit-Learn processing across clusters of machines. These packages deserve entire books about them. This is not one of those books. But no overview of Python sequences would be complete without at least a quick look at NumPy arrays.

Having looked at flat sequences—standard arrays and NumPy arrays—we now turn to a completely different set of replacements for the plain old `list`: queues.

## Dequeues and Other Queues

The `.append` and `.pop` methods make a `list` usable as a stack or a queue (if you use `.append` and `.pop(0)`, you get FIFO behavior). But inserting and removing from the head of a list (the 0-index end) is costly because the entire list must be shifted in memory.

The class `collections.deque` is a thread-safe double-ended queue designed for fast inserting and removing from both ends. It is also the way to go if you need to keep a list of “last seen items” or something like that, because a `deque` can be bounded—i.e., created with a fixed maximum length—and then, when it is full, it discards items from the opposite end when you add new ones. [Example 2-23](#) shows some typical operations performed on a `deque`.

### *Example 2-23. Working with a deque*

---

```
>>> from collections import deque
>>> dq = deque(range(10), maxlen=10) ❶
>>> dq
deque([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], maxlen=10)
>>> dq.rotate(3) ❷
>>> dq
deque([7, 8, 9, 0, 1, 2, 3, 4, 5, 6], maxlen=10)
>>> dq.rotate(-4)
>>> dq
deque([1, 2, 3, 4, 5, 6, 7, 8, 9, 0], maxlen=10)
>>> dq.appendleft(-1) ❸
>>> dq
deque([-1, 1, 2, 3, 4, 5, 6, 7, 8, 9], maxlen=10)
>>> dq.extend([11, 22, 33]) ❹
>>> dq
deque([3, 4, 5, 6, 7, 8, 9, 11, 22, 33], maxlen=10)
>>> dq.extendleft([10, 20, 30, 40]) ❺
>>> dq
deque([40, 30, 20, 10, 3, 4, 5, 6, 7, 8], maxlen=10)
```

- ❶ The optional `maxlen` argument sets the maximum number of items allowed in this instance of `deque`; this sets a read-only

`maxlen` instance attribute.

- ② Rotating with `n > 0` takes items from the right end and prepends them to the left; when `n < 0` items are taken from left and appended to the right.
- ③ Appending to a deque that is full (`len(d) == d.maxlen`) discards items from the other end; note in the next line that the `0` is dropped.
- ④ Adding three items to the right pushes out the leftmost `-1`, `1`, and `2`.
- ⑤ Note that `extendleft(iter)` works by appending each successive item of the `iter` argument to the left of the deque, therefore the final position of the items is reversed.

Table 2-3 compares the methods that are specific to `list` and `deque` (removing those that also appear in `object`).

Note that `deque` implements most of the `list` methods, and adds a few specific to its design, like `popleft` and `rotate`. But there is a hidden cost: removing items from the middle of a `deque` is not as fast. It is really optimized for appending and popping from the ends.

The `append` and `popleft` operations are atomic, so `deque` is safe to use as a FIFO queue in multithreaded applications without the need for using locks.

*Table 2-3. Methods implemented in list or deque (those that are also implemented by object were omitted for brevity)*

	list	deque	
<code>s.__add__(s2)</code>	•		<code>s + s2</code> —concatenation
<code>s.__iadd__(s2)</code>	•	•	<code>s += s2</code> —in-place concatenation
<code>s.append(e)</code>	•	•	Append one element to the right (after last)
<code>s.appendleft(e)</code>		•	Append one element to the left (before first)
<code>s.clear()</code>	•	•	Delete all items
<code>s.__contains__(e)</code>	•		<code>e in s</code>
<code>s.copy()</code>	•		Shallow copy of the list
<code>s.__copy__()</code>		•	Support for <code>copy.copy</code> (shallow copy)
<code>s.count(e)</code>	•	•	Count occurrences of an element
<code>s.__delitem__(p)</code>	•	•	Remove item at position <code>p</code>
<code>s.extend(i)</code>	•	•	Append items from iterable <code>i</code> to the right
<code>s.extendleft(i)</code>		•	Append items from iterable <code>i</code> to the left
<code>s.__getitem__(p)</code>	•	•	<code>s[p]</code> —get item at position
<code>s.index(e)</code>	•		Find position of first occurrence of <code>e</code>
<code>s.insert(p, e)</code>	•		Insert element <code>e</code> before the item at position <code>p</code>
<code>s.__iter__()</code>	•	•	Get iterator
<code>s.__len__()</code>	•	•	<code>len(s)</code> —number of items



	list	deque	
<code>s.__mul__(n)</code>	•		<code>s * n</code> —repeated concatenation
<code>s.__imul__(n)</code>	•		<code>s *= n</code> —in-place repeated concatenation
<code>s.__rmul__(n)</code>	•		<code>n * s</code> —reversed repeated concatenation <sup>a</sup>
<code>s.pop()</code>	•	•	Remove and return last item <sup>b</sup>
<code>s.popleft()</code>		•	Remove and return first item
<code>s.remove(e)</code>	•	•	Remove first occurrence of element <code>e</code> by value
<code>s.reverse()</code>	•	•	Reverse the order of the items in place
<code>s.__reversed__()</code>	•	•	Get iterator to scan items from last to first
<code>s.rotate(n)</code>		•	Move <code>n</code> items from one end to the other
<code>s.__setitem__(p, e)</code>	•	•	<code>s[p] = e</code> —put <code>e</code> in position <code>p</code> , overwriting existing item
<code>s.sort([key], [reverse])</code>	•		Sort items in place with optional keyword arguments <code>key</code> and <code>reverse</code>

<sup>a</sup> Reversed operators are explained in [Link to Come].

<sup>b</sup> `a_list.pop(p)` allows removing from position `p` but `deque` does not support that option.

Besides `deque`, other Python standard library packages implement queues:

### *queue*

This provides the synchronized (i.e., thread-safe) classes `SimpleQueue`, `Queue`, `LifoQueue`, and `PriorityQueue`. These

can be used for safe communication between threads. All except `SimpleQueue` can be bounded by providing a `maxsize` argument greater than 0 to the constructor. However, they don't discard items to make room as `deque` does. Instead, when the queue is full the insertion of a new item blocks—i.e., it waits until some other thread makes room by taking an item from the queue, which is useful to throttle the number of live threads.

### *multiprocessing*

Implements its own unbounded `SimpleQueue` and bounded `Queue`, very similar to those in the `queue` package, but designed for interprocess communication. A specialized `multiprocessing.JoinableQueue` is also available for easier task management.

### *asyncio*

Provides `Queue`, `LifoQueue`, `PriorityQueue`, and `JoinableQueue` with APIs inspired by the classes in the `queue` and `multiprocessing` modules, but adapted for managing tasks in asynchronous programming.

### *heapq*

In contrast to the previous three modules, `heapq` does not implement a queue class, but provides functions like `heappush` and `heappop` that let you use a mutable sequence as a heap queue or priority queue.

This ends our overview of alternatives to the `list` type, and also our exploration of sequence types in general—except for the particulars of `str` and binary sequences, which have their own chapter (Chapter 4).

## Chapter Summary

Mastering the standard library sequence types is a prerequisite for writing concise, effective, and idiomatic Python code.

Python sequences are often categorized as mutable or immutable, but it is also useful to consider a different axis: flat sequences and container sequences. The former are more compact, faster, and easier to use, but are limited to storing atomic data such as numbers, characters, and bytes. Container sequences are more flexible, but may surprise you when they hold mutable objects, so you need to be careful to use them correctly with nested data structures.

Unfortunately, Python has no foolproof immutable container sequence type: even “immutable” tuples can have their values change, when they contain mutable items like lists or user-defined objects.

List comprehensions and generator expressions are powerful notations to build and initialize sequences. If you are not yet comfortable with them, take the time to master their basic usage. It is not hard, and soon you will be hooked.

Tuples in Python play two roles: as records with unnamed fields and as immutable lists. When a tuple is used as a record, tuple unpacking is the safest, most readable way of getting at the fields. The new `*` syntax makes tuple unpacking even better by making it easier to ignore some fields and to deal with optional fields. When using a tuple as an immutable list, remember that a tuple value is only guaranteed to be fixed if all the items in it are also immutable. Calling `hash(t)` on a tuple is a quick way to assert that its value is fixed. A `TypeError` will be raised if `t` contains mutable items.

Sequence slicing is a favorite Python syntax feature, and it is even more powerful than many realize. Multidimensional slicing and

ellipsis (`...`) notation, as used in NumPy, may also be supported by user-defined sequences. Assigning to slices is a very expressive way of editing mutable sequences.

Repeated concatenation as in `seq*n` is convenient and, with care, can be used to initialize lists of lists containing immutable items. Augmented assignment with `+=` and `*=` behaves differently for mutable and immutable sequences. In the latter case, these operators necessarily build new sequences. But if the target sequence is mutable, it is usually changed in place—but not always, depending on how the sequence is implemented.

The `sort` method and the `sorted` built-in function are easy to use and flexible, thanks to the `key` optional argument they accept, with a function to calculate the ordering criterion. By the way, `key` can also be used with the `min` and `max` built-in functions. To keep a sorted sequence in order, always insert items into it using `bisect.insort`; to search it efficiently, use `bisect.bisect`.

Beyond lists and tuples, the Python standard library provides `array.array`. Although NumPy and SciPy are not part of the standard library, if you do any kind of numerical processing on large sets of data, studying even a small part of these libraries can take you a long way.

We closed by visiting the versatile and thread-safe `collections.deque`, comparing its API with that of `list` in [Table 2-3](#) and mentioning other queue implementations in the standard library.

## Further Reading

Chapter 1, “Data Structures” of *Python Cookbook, 3rd Edition* (O’Reilly) by David Beazley and Brian K. Jones has many recipes focusing on sequences, including “Recipe 1.11. Naming a Slice,”

from which I learned the trick of assigning slices to variables to improve readability, illustrated in our [Example 2-9](#).

The second edition of *Python Cookbook* was written for Python 2.4, but much of its code works with Python 3, and a lot of the recipes in Chapters 5 and 6 deal with sequences. The book was edited by Alex Martelli, Anna Martelli Ravenscroft, and David Ascher, and it includes contributions by dozens of Pythonistas. The third edition was rewritten from scratch, and focuses more on the semantics of the language—particularly what has changed in Python 3—while the older volume emphasizes pragmatics (i.e., how to apply the language to real-world problems). Even though some of the second edition solutions are no longer the best approach, I honestly think it is worthwhile to have both editions of *Python Cookbook* on hand.

The official Python [Sorting HOW TO](#) has several examples of advanced tricks for using `sorted` and `list.sort`.

[PEP 3132 — Extended Iterable Unpacking](#) is the canonical source to read about the new use of `*extra` syntax on the left hand of parallel assignments.

If you'd like a glimpse of Python evolving, [Missing \\*-unpacking generalizations](#) is a bug tracker that proposed enhancements to the iterable unpacking notation. [PEP 448 — Additional Unpacking Generalizations](#) resulted from the discussions in that issue. The proposed were merged into Python 3.5, after the first edition of this book was printed.

Raymond Hettinger's response to [Are tuples more efficient than lists in Python?](#) on StackOverflow is great, but does not mention that `tuple(t)` returns `t` only if it is a hashable tuple. When `t` is unhashable, `tuple(t)` returns a shallow copy of `t`. Also, implementation details may have changed since that answer was posted in 2014. In particular, I found much smaller space savings

with `sys.getsizeof` using the same example tuple and list objects as Hettinger did. Artem Golubin wrote a blog post on the same subject, titled [Optimization tricks in Python: lists and tuples](#), last updated in April, 2018 as I write this.

Eli Bendersky's blog post [“Less Copies in Python with the Buffer Protocol and memoryviews”](#) includes a short tutorial on `memoryview`.

There are numerous books covering NumPy in the market, and many don't mention “NumPy” in the title. Two examples are the open access [Python Data Science Handbook](#) by Jake VanderPlas, and Wes McKinney's [Python for Data Analysis, 2e](#).

“NumPy is all about vectorization”. That is the opening sentence of Nicolas P. Rougier's open access book [From Python to NumPy](#). Vectorized operations apply mathematical functions to all elements of an array without an explicit loop written in Python. They can operate in parallel, using special vector instructions in modern CPUs, leveraging multiple cores or delegating to the GPU, depending on the library. The first example in Rougier's book shows a speedup of 500 times after refactoring a nice Pythonic class using a generator method, into a lean and mean function calling a couple of NumPy vector functions.

Fernando Perez—a physicist—created IPython, an incredibly powerful replacement for the Python console that also provides a GUI, integrated inline graph plotting, literate programming support (interleaving text with code), and rendering to PDF. Interactive, multimedia IPython sessions can be shared over HTTP as Jupyter notebooks. See screenshots and video at [The IPython Notebook](#). IPython was so successful that the project received millions of dollars in grants from the Sloan Foundation and other research institutions to hire full-time developers to enhance it. That effort resulted in [Project Jupyter](#) and the new JupyterLab web-based IDE released in July, 2019. Scientists love the combination of Python's elegant power,

Jupyter's interactivity, and the strengths of NumPy/SciPy. That's what made Python one of the top languages in scientific computing, paving the way for its success in the field of Machine Learning.

To learn how to use `deque` (and other collections) see the examples and practical recipes in [8.3. collections — Container datatypes](#) in the Python documentation.

The best defense of the Python convention of excluding the last item in ranges and slices was written by Edsger W. Dijkstra himself, in a short memo titled [“Why Numbering Should Start at Zero”](#). The subject of the memo is mathematical notation, but it's relevant to Python because Dijkstra explains with rigor and humor why a sequence like 2, 3, ..., 12 should always be expressed as  $2 \leq i < 13$ . All other reasonable conventions are refuted, as is the idea of letting each user choose a convention. The title refers to zero-based indexing, but the memo is really about why it is desirable that `'ABCDE'[1:3]` means `'BC'` and not `'BCD'` and why it makes perfect sense to write `range(2, 13)` to produce 2, 3, 4, ..., 12. By the way, the memo is a handwritten note, but it's beautiful and totally readable. Dijkstra handwriting is so clear that someone created a [font](#) out of his notes.

## SOAPBOX

### The Nature of Tuples

In 2012, I presented a poster about the ABC language at PyCon US. Before creating Python, Guido had worked on the ABC interpreter, so he came to see my poster. Among other things, we talked about the ABC *compounds*, which are clearly the predecessors of Python tuples. Compounds also support parallel assignment and are used as composite keys in dictionaries (or *tables*, in ABC parlance). However, compounds are not sequences. They are not iterable and you cannot retrieve a field by index, much less slice them. You either handle the compound as whole or extract the individual fields using parallel assignment, that's all.

I told Guido that these limitations make the main purpose of compounds very clear: they are just records without field names. His response: "Making tuples behave as sequences was a hack."

This illustrates the pragmatic approach that makes Python so much better and more successful than ABC. From a language implementer perspective, making tuples behave as sequences costs little. As a result, tuples may not be as "conceptually pure" as compounds, but we have many more ways of using them. They can even be used as immutable lists, of all things!

It is really useful to have immutable lists in the language, even if their type is not called `frozenset` but is really `tuple` behaving as a sequence.

### "Elegance Begets Simplicity"

The use of the syntax `*extra` to assign multiple items to a parameter started with function definitions a long time ago (I have a book about Python 1.4 from 1996 that covers that). Starting with Python 1.6, the form `*extra` can be used in the context of function calls to unpack an iterable into multiple arguments, a complementary operation. This is elegant, makes intuitive sense, and made the `apply` function redundant (it's now gone). Now, with Python 3, the `*extra` notation also works on the left of parallel assignments to grab excess items, enhancing what was already a handy language feature.

With each of these changes, Python became more flexible, more consistent, and simpler. In one word: it became more elegant. "Elegance begets simplicity" is the motto on my favorite PyCon T-shirt from Chicago, 2009. It is decorated with a painting by Bruce Eckel depicting ䷡—hexagram 22 of the I Ching, 賁 (bì), "Adorning," sometimes translated as "Grace" or "Beauty."

### Flat Versus Container Sequences



To highlight the different memory models of the sequence types, I used the terms *container sequence* and *flat sequence*. The “container” word is from the [Data Model documentation](#):

*Some objects contain references to other objects; these are called containers.*

I used the term “container sequence” to be specific, because there are containers in Python that are not sequences, like `dict` and `set`. Container sequences can be nested because they may contain objects of any type, including their own type.

On the other hand, *flat sequences* are sequence types that cannot be nested because they only hold simple atomic types like integers, floats, or characters.

I adopted the term *flat sequence* because I needed something to contrast with “container sequence.”

**Update:** despite the previous use of the word “containers” in the official documentation, there is an abstract class in `collections.abc` called `Container`. That ABC has just one method, `__contains__`—the special method behind the `in` operator. This means that strings and arrays, which are not containers in the traditional sense, are virtual subclasses of `Container` because they implement `__contains__`. This is unfortunate but it happened. It’s just one more example of humans using a word to mean different things. In this book I’ll write “container” with lowercase letters to mean “an object that contains references to other objects” and `Container` with capitalized initial in a single-spaced font to refer to `collections.abc.Container` or classes that implement `__contains__`.

## Mixed Bag Lists

Introductory Python texts emphasize that lists can contain objects of mixed types, but in practice that feature is not very useful: we put items in a list to process them later, which implies that all items should support at least some operation in common (i.e., they should all “quack” whether or not they are genetically 100% ducks). For example, you can’t sort a list in Python 3 unless the items in it are comparable:

```
>>> l = [28, 14, '28', 5, '9', '1', 0, 6, '23', 19]
>>> sorted(l)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: str() < int()
```

Unlike lists, tuples often hold items of different types. That is natural, considering that each item in a tuple is really a field, and each field type is

independent of the others.

## Key Is Brilliant

The key optional argument of `list.sort`, `sorted`, `max`, and `min` is a great idea. Other languages force you to provide a two-argument comparison function like the deprecated `cmp(a, b)` function in Python 2. Using `key` is both simpler and more efficient. It's simpler because you just define a one-argument function that retrieves or calculates whatever criterion you want to use to sort your objects; this is easier than writing a two-argument function to return `-1`, `0`, `1`. It is also more efficient because the key function is invoked only once per item, while the two-argument comparison is called every time the sorting algorithm needs to compare two items. Of course, Python also has to compare the keys while sorting, but that comparison is done in optimized C code and not in a Python function that you wrote.

By the way, using `key` actually lets us sort a mixed bag of numbers and number-like strings. You just need to decide whether you want to treat all items as integers or strings:

```
>>> l = [28, 14, '28', 5, '9', '1', 0, 6, '23', 19]
>>> sorted(l, key=int)
[0, '1', 5, 6, '9', 14, 19, '23', 28, '28']
>>> sorted(l, key=str)
[0, '1', 14, 19, '23', 28, '28', 5, 6, '9']
```

## Oracle, Google, and the Timbot Conspiracy

The sorting algorithm used in `sorted` and `list.sort` is Timsort, an adaptive algorithm that switches from insertion sort to merge sort strategies, depending on how ordered the data is. This is efficient because real-world data tends to have runs of sorted items. There is a [Wikipedia article](#) about it.

Timsort was first deployed in CPython, in 2002. Since 2009, Timsort is also used to sort arrays in both standard Java and Android, a fact that became widely known when Oracle used some of the code related to Timsort as evidence of Google infringement of Sun's intellectual property. See [Oracle v. Google - Day 14 Filings](#).

Timsort was invented by Tim Peters, a Python core developer so prolific that he is believed to be an AI, the Timbot. You can read about that conspiracy theory in [Python Humor](#). Tim also wrote The Zen of Python: `import this`.

- 
- 1 Leo Geurts, Lambert Meertens, and Steven Pemberton, *ABC Programmer's Handbook*, p. 8.
  - 2 In “Memory Views” we show that especially constructed memory views can have more than one dimension.
  - 3 No, I did not get this backwards: the `ellipsis` class name is really all lowercase and the instance is a built-in named `Ellipsis`, just like `bool` is lowercase but its instances are `True` and `False`.
  - 4 `str` is an exception to this description. Because string building with `+=` in loops is so common in the wild, CPython is optimized for this use case. `str` instances are allocated in memory with room to spare, so that concatenation does not require copying the whole string every time.
  - 5 Thanks to Leonardo Rochaël and Cesar Kawakami for sharing this riddle at the 2013 PythonBrasil Conference.
  - 6 A reader suggested that the operation in the example can be done with `t[2].extend([50,60])`, without errors. I am aware of that, but my intent is to show the strange behavior of the `+=` operator in this case.
  - 7 Receiver is the target of a method call, the object bound to `self` in the method body.
  - 8 The examples also demonstrate that Timsort—the sorting algorithm used in Python—is stable (i.e., it preserves the relative ordering of items that compare equal). Timsort is discussed further in the “Soapbox” sidebar at the end of this chapter.
  - 9 The words in this example are sorted alphabetically because they are 100% made of lowercase ASCII characters. See warning after the example.
  - 10 Thanks to reader Gregory Sherman for pointing this out.

## Chapter 3. Dictionaries and Sets

---

### A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at [fluentpython2e@ramalho.org](mailto:fluentpython2e@ramalho.org).

*May your hashes be unique,  
Your keys rarely collide,  
And your dictionaries  
be forever ordered.<sup>1</sup>*

—Brandon Rhodes, in *The Dictionary Even Mightier*

The `dict` type is not only widely used in our programs but also a fundamental part of the Python implementation. Class and instance attributes, module namespaces, and function keyword arguments are some of the fundamental Python constructs represented by dictionaries in memory. The built-in functions are all in `__builtins__.__dict__`.

Because of their crucial role, Python dicts are highly optimized—and continue to get improvements. *Hash tables* are the engines behind Python's high-performance dicts.

Other built-in types based on hash tables are `set` and `frozenset`. These offer richer APIs and operators than the sets you may have encountered in other popular languages. In particular, Python sets implement all the fundamental operations from set theory, like union, intersection, subset tests etc. With them, we can express algorithms in a more declarative way, avoiding lots of nested loops and conditionals.

Here is a brief outline of this chapter:

- Common dictionary methods
- Special handling for missing keys
- Variations of `dict` in the standard library
- The `set` and `frozenset` types

- How hash tables work
- Implications of hash tables in the behavior of sets and dictionaries.

## What's new in this chapter

The `dict` implementation evolved from what I described in 1<sup>st</sup> edition of *Fluent Python*. Major revisions in this chapter are:

- Explanation of the hash table algorithm now starts with its use in `set`, which is simpler to understand.
- Coverage of the memory optimizations that preserve key insertion order in `dict` instances—implemented in Python 3.6—and the key-sharing layout for dictionaries holding instance attributes—the `__dict__` of used-defined objects since Python 3.3.
- New section on the view objects returned by `dict.keys`, `dict.items`, and `dict.values` since Python 3.0.

### NOTE

One minor change: I adopted the term *hash code* instead of *hash value*, because we need to talk about object values to understand hashing, and it is easier to keep the two concepts apart in a sentence like this: “The *hash code* is derived from the *object value*.” However, I keep the original term when I quote the Python documentation.

## Standard API of Mapping Types

The `collections.abc` module provides the `Mapping` and `MutableMapping` ABCs describing the interfaces of `dict` and similar types. See Figure 3-1.

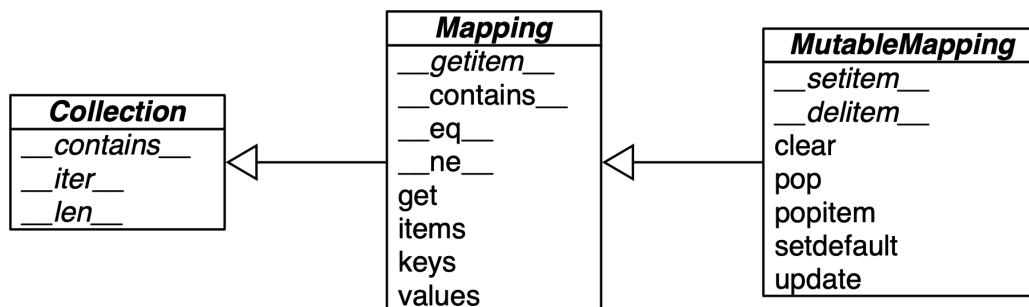


Figure 3-1. Simplified UML class diagram for the `MutableMapping` and its superclasses from `collections.abc` (inheritance arrows point from subclasses to superclasses; names in *italic* are abstract classes and abstract methods)

The main value of the ABCs is documenting and formalizing the standard interfaces for mappings, and serving as criteria for `isinstance` tests in code that needs to support mappings in a broad sense:

```
>>> my_dict = {}
>>> isinstance(my_dict, abc.Mapping)
True
>>> isinstance(my_dict, abc.MutableMapping)
True
```

#### TIP

Using `isinstance` with an ABC is often better than checking whether a function argument is of the concrete `dict` type, because then alternative mapping types can be used. We'll discuss this in detail in [\[Link to Come\]](#).

To implement a custom mapping, it's easier to extend `collections.UserDict`, or to wrap a `dict` by composition, instead of subclassing these ABCs. The `collections.UserDict` class and all concrete mapping classes in the standard library encapsulate the basic `dict` in their implementation, which in turn is built on a hash table. Therefore, they all share the limitation that the keys must be *hashable* (the values need not be hashable, only the keys). If you need a refresher, check out [“What Is Hashable?”](#).

## WHAT IS HASHABLE?

Here is part of the definition of hashable from the [Python Glossary](#):

*An object is hashable if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` method). Hashable objects which compare equal must have the same hash value. [...]*

Numeric types and flat immutable types `str` and `bytes` are all hashable. Container types are hashable if they are immutable and all contained objects are also hashable. A `frozenset` is always hashable, because every element it contains must be hashable by definition. A `tuple` is hashable only if all its items are hashable. See tuples `tt`, `tl`, and `tf`:

```
>>> tt = (1, 2, (30, 40))
>>> hash(tt)
8027212646858338501
>>> tl = (1, 2, [30, 40])
>>> hash(tl)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> tf = (1, 2, frozenset([30, 40]))
>>> hash(tf)
-4118419923444501110
```

User-defined types are hashable by default because their hash code is their `id()` and the `__eq__()` method inherited from the object class simply compares the object ids. If an object implements a custom `__eq__()` which takes into account its internal state, it will be hashable only if its `__hash__()` always returns the same hash code. In practice, this requires that `__eq__()` and `__hash__()` only take into account instance attributes that never change during the life of the object.

Given these ground rules, you can build dictionaries in several ways. The [Built-in Types](#) page in the Library Reference has this example to show the various means of building a dict:

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'three': 3, 'two': 2, 'one': 1}
>>> c = dict([('two', 2), ('one', 1), ('three', 3)])
>>> d = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> a == b == c == d == e
True
```

Note that all of those `dict` instances are considered equal because they have the same set of keys and values, even if the order of the keys is not the same.

CPython 3.6 started preserving the insertion order of the keys as an implementation detail, and Guido van Rossum declared it an official language feature in Python 3.7, so we can depend on it:

```

>>> a
{'one': 1, 'two': 2, 'three': 3}
>>> list(a.keys())
['one', 'two', 'three']
>>> c
{'two': 2, 'one': 1, 'three': 3}
>>> c.popitem()
('three', 3)
>>> c
{'two': 2, 'one': 1}

```

Before Python 3.6, `c.popitem()` would remove and return an arbitrary key-value pair. Now it always removes and returns the last key-value pair added to the dict.

In addition to the literal syntax and the flexible `dict` constructor, we can use *dict comprehensions* to build dictionaries. See the next section.

## dict Comprehensions

Since Python 2.7, the syntax of listcomps and genexps was adapted to `dict` comprehensions (and `set` comprehensions as well, which we'll soon visit). A *dictcomp* builds a `dict` instance by taking `key:value` pairs from any iterable. [Example 3-1](#) shows the use of `dict` comprehensions to build two dictionaries from the same list of tuples.

*Example 3-1. Examples of dict comprehensions*

```

>>> dial_codes = [
...     (880, 'Bangladesh'),
...     (55, 'Brazil'),
...     (86, 'China'),
...     (91, 'India'),
...     (62, 'Indonesia'),
...     (81, 'Japan'),
...     (234, 'Nigeria'),
...     (92, 'Pakistan'),
...     (7, 'Russia'),
...     (1, 'United States'),
... ]
>>> country_dial = {country: code for code, country in dial_codes}
>>> country_dial
{'Bangladesh': 880, 'Brazil': 55, 'China': 86, 'India': 91, 'Indonesia': 62,
'Japan': 81, 'Nigeria': 234, 'Pakistan': 92, 'Russia': 7, 'United States': 1}
>>> {code: country.upper()
...     for country, code in sorted(country_dial.items())
...     if code < 70}
{55: 'BRAZIL', 62: 'INDONESIA', 7: 'RUSSIA', 1: 'UNITED STATES'}

```



- ❶ An iterable of key-value pairs like `dia1_codes` can be passed directly to the `dict` constructor, but...
- ❷ ...here we swap the pairs: `country` is the key, and `code` is the value.s
- ❸ Sorting `country_dia1` by name, reversing the pairs again, uppercasing values, and filtering items by `code < 66`.

If you're used to listcomps, dictcomps are a natural next step. If you aren't, the spread of the comprehension syntax means it's now more profitable than ever to become fluent in it.

We now move to a panoramic view of the API for mappings.

## Overview of Common Mapping Methods

The basic API for mappings is quite rich. [Table 3-1](#) shows the methods implemented by `dict` and two of its most useful variations: `defaultdict` and `OrderedDict`, both defined in the `collections` module.

*Table 3-1. Methods of the mapping types dict, collections.defaultdict, and collections.OrderedDict (common object methods omitted for brevity); optional arguments are enclosed in [...]*

	dict	defaultdict	OrderedDict	
d.clear()	•	•	•	Remove all items
d.__contains__(k)	•	•	•	k in d
d.copy()	•	•	•	Shallow copy
d.__copy__()		•		Support for copy.copy
d.default_factory		•		Callable invoked by __missing__ to set missing values <sup>a</sup>
d.__delitem__(k)	•	•	•	del d[k]—remove item with key k
d.fromkeys(it, [initial])	•	•	•	New mapping from keys in iterable, with optional initial value (defaults to None)
d.get(k, [default])	•	•	•	Get item with key k, return default or None if missing
d.__getitem__(k)	•	•	•	d[k]—get item with key k
d.items()	•	•	•	Get view over items—(key, value) pairs
d.__iter__()	•	•	•	Get iterator over keys
d.keys()	•	•	•	Get view over keys
d.__len__()	•	•	•	len(d)—number of items
d.__missing__(k)		•		Called when __getitem__ cannot find the key
d.move_to_end(k, [last])			•	Move k first or last position (last is True by default)
d.pop(k, [default])	•	•	•	Remove and return value at k, or default or None if missing
d.popitem()	•	•	•	Remove and return the last inserted item as (key, value) <sup>b</sup>
d.__reversed__()	•	•	•	Get iterator for keys from last to first inserted

	dict	defaultdict	OrderedDict	
<code>d.setdefault(k, [default])</code>	•	•	•	If <code>k</code> in <code>d</code> , return <code>d[k]</code> ; else set <code>d[k] = default</code> and return it
<code>d.__setitem__(k, v)</code>	•	•	•	<code>d[k] = v</code> —put <code>v</code> at <code>k</code>
<code>d.update(m, /**k args])</code>	•	•	•	Update <code>d</code> with items from mapping or iterable of ( <code>key</code> , <code>value</code> ) pairs
<code>d.values()</code>	•	•	•	Get <i>view</i> over values

a `default_factory` is not a method, but a callable attribute set by the end user when a `defaultdict` is instantiated.

b `OrderedDict.popitem(last=False)` removes the first item inserted (FIFO). This keyword argument is not valid for `dict` or `defaultdict` as of Python 3.8.

The way `d.update(m)` handles its first argument `m` is a prime example of *duck typing*: it first checks whether `m` has a `keys` method and, if it does, assumes it is a mapping. Otherwise, `update()` falls back to iterating over `m`, assuming its items are (`key`, `value`) pairs. The constructor for most Python mappings uses the logic of `update()` internally, which means they can be initialized from other mappings or from any iterable object producing (`key`, `value`) pairs.

A subtle mapping method is `setdefault()`. It avoids redundant key lookups when the value of a dictionary item is mutable and we need to update it in-place. If you are not comfortable using it, the following section explains how, through a practical example.

## Handling Missing Keys with `setdefault`

In line with Python’s *fail-fast* philosophy, `dict` access with `d[k]` raises an error when `k` is not an existing key. Every Pythonista knows that `d.get(k, default)` is an alternative to `d[k]` whenever a default value is more convenient than handling `KeyError`. However, when updating the mutable value found, using either `d[k]` or `get` is awkward and inefficient.

Consider a script to index text, producing a mapping where each key is a word and the value is a list of positions where that word occurs, as shown in [Example 3-2](#).

*Example 3-2. Partial output from [Example 3-3](#) processing the Zen of Python; each line shows a word and a list of occurrences coded as pairs: (line\_number, column\_number)*

---

```

$ python3 index0.py zen.txt
a [(19, 48), (20, 53)]
Although [(11, 1), (16, 1), (18, 1)]
ambiguity [(14, 16)]
and [(15, 23)]
are [(21, 12)]
aren [(10, 15)]
at [(16, 38)]
bad [(19, 50)]
be [(15, 14), (16, 27), (20, 50)]
beats [(11, 23)]
Beautiful [(3, 1)]
better [(3, 14), (4, 13), (5, 11), (6, 12), (7, 9), (8, 11),
(17, 8), (18, 25)]
...

```

Example 3-3, a suboptimal script written to show one case where `dict.get` is not the best way to handle a missing key.

I adapted it from an example by Alex Martelli,<sup>2</sup>.

*Example 3-3. `index0.py` uses `dict.get` to fetch and update a list of word occurrences from the index (a better solution is in Example 3-4)*

```

"""Build an index mapping word -> list of occurrences"""

import sys
import re

WORD_RE = re.compile(r'\w+')

index = {}
with open(sys.argv[1], encoding='utf-8') as fp:
    for line_no, line in enumerate(fp, 1):
        for match in WORD_RE.finditer(line):
            word = match.group()
            column_no = match.start()+1
            location = (line_no, column_no)
            # this is ugly; coded like this to make a point
            occurrences = index.get(word, []) ❶
            occurrences.append(location)       ❷
            index[word] = occurrences         ❸

# print in alphabetical order
for word in sorted(index, key=str.upper): ❹
    print(word, index[word])

```

- ❶ Get the list of occurrences for word, or `[]` if not found.
- ❷ Append new location to occurrences.
- ❸

Put changed occurrences into `index` dict; this entails a second search through the `index`.

- ④ In the `key=` argument of `sorted` I am not calling `str.upper`, just passing a reference to that method so the `sorted` function can use it to normalize the words for sorting.<sup>3</sup>

The three lines dealing with occurrences in [Example 3-3](#) can be replaced by a single line using `dict.setdefault`. [Example 3-4](#) is closer to Alex Martelli's original example.

*Example 3-4. `index.py` uses `dict.setdefault` to fetch and update a list of word occurrences from the index in a single line; contrast with [Example 3-3](#)*

```
"""Build an index mapping word -> list of occurrences"""

import sys
import re

WORD_RE = re.compile(r'\w+')

index = {}
with open(sys.argv[1], encoding='utf-8') as fp:
    for line_no, line in enumerate(fp, 1):
        for match in WORD_RE.finditer(line):
            word = match.group()
            column_no = match.start()+1
            location = (line_no, column_no)
            index.setdefault(word, []).append(location) ❶

# print in alphabetical order
for word in sorted(index, key=str.upper):
    print(word, index[word])
```

- ❶ Get the list of occurrences for `word`, or set it to `[]` if not found; `setdefault` returns the value, so it can be updated without requiring a second search.

In other words, the end result of this line...

```
my_dict.setdefault(key, []).append(new_value)
```

...is the same as running...

```
if key not in my_dict:
    my_dict[key] = []
my_dict[key].append(new_value)
```

...except that the latter code performs at least two searches for `key`—three if it's not found—while `setdefault` does it all with a single lookup.

A related issue, handling missing keys on any lookup (and not only when inserting), is the subject of the next section.

## Mappings with Flexible Key Lookup

Sometimes it is convenient to have mappings that return some made-up value when a missing key is searched. There are two main approaches to this: one is to use a `defaultdict` instead of a plain `dict`. The other is to subclass `dict` or any other mapping type and add a `__missing__` method. Both solutions are covered next.

### `defaultdict`: Another Take on Missing Keys

[Example 3-5](#) uses `collections.defaultdict` to provide another elegant solution to the problem in [Example 3-4](#). A `defaultdict` is configured to create items on demand whenever a missing key is searched.

Here is how it works: when instantiating a `defaultdict`, you provide a callable that is used to produce a default value whenever `__getitem__` is passed a nonexistent key argument.

For example, given an empty `defaultdict` created as `dd = defaultdict(list)`, if 'new-key' is not in `dd`, the expression `dd['new-key']` does the following steps:

1. Calls `list()` to create a new list.
2. Inserts the list into `dd` using 'new-key' as key.
3. Returns a reference to that list.

The callable that produces the default values is held in an instance attribute called `default_factory`.

*Example 3-5. `index_default.py`: using an instance of `defaultdict` instead of the `setdefault` method*

---

```
"""Build an index mapping word -> list of occurrences"""
```

```
import sys
import re
import collections

WORD_RE = re.compile(r'\w+')

index = collections.defaultdict(list)
with open(sys.argv[1], encoding='utf-8') as fp:
    for line_no, line in enumerate(fp, 1):
        for match in WORD_RE.finditer(line):
            word = match.group()
```

```

        column_no = match.start()+1
        location = (line_no, column_no)
        index[word].append(location) ❷

# print in alphabetical order
for word in sorted(index, key=str.upper):
    print(word, index[word])

```

- ❶ Create a defaultdict with the list constructor as default\_factory.
- ❷ If word is not initially in the index, the default\_factory is called to produce the missing value, which in this case is an empty list that is then assigned to index[word] and returned, so the .append(location) operation always succeeds.

If no default\_factory is provided, the usual KeyError is raised for missing keys.

### WARNING

The default\_factory of a defaultdict is only invoked to provide default values for `__getitem__` calls, and not for the other methods. For example, if `dd` is a defaultdict, and `k` is a missing key, `dd[k]` will call the default\_factory to create a default value, but `dd.get(k)` still returns `None`.

The mechanism that makes defaultdict work by calling default\_factory is the `__missing__` special method, a feature that we discuss next.

## The `__missing__` Method

Underlying the way mappings deal with missing keys is the aptly named `__missing__` method. This method is not defined in the base dict class, but dict is aware of it: if you subclass dict and provide a `__missing__` method, the standard dict.`__getitem__` will call it whenever a key is not found, instead of raising `KeyError`.

### WARNING

The `__missing__` method is only called by `__getitem__` (i.e., for the `d[k]` operator). The presence of a `__missing__` method has no effect on the behavior of other methods that look up keys, such as `get` or `__contains__` (which implements the `in` operator). This is why the default\_factory of defaultdict works only with `__getitem__`, as noted in the warning at the end of the previous section.

Suppose you'd like a mapping where keys are converted to `str` when looked up. A concrete use case is a device library for IoT<sup>4</sup>, where a programmable board with general purpose I/O pins (e.g., a Raspberry Pi or an Arduino) is represented by a `Board` class with a `my_board.pins` attribute, which is a mapping of physical pin identifiers to pin software objects. The physical pin identifier may be just a number or a string like "A0" or "P9\_12". For consistency, it is desirable that all keys in `board.pins` are strings, but it is also convenient that looking up a pin by number, as in `my_arduino.pin[13]`, so that beginners are not tripped when they want to blink the LED on pin 13 of their Arduinos. [Example 3-6](#) shows how such a mapping would work.

*Example 3-6. When searching for a nonstring key, `StrKeyDict0` converts it to `str` when it is not found*

Tests for item retrieval using `d[key]` notation::

```
>>> d = StrKeyDict0([('2', 'two'), ('4', 'four')])
>>> d['2']
'two'
>>> d[4]
'four'
>>> d[1]
Traceback (most recent call last):
...
KeyError: '1'
```

Tests for item retrieval using `d.get(key)` notation::

```
>>> d.get('2')
'two'
>>> d.get(4)
'four'
>>> d.get(1, 'N/A')
'N/A'
```

Tests for the `in` operator::

```
>>> 2 in d
True
>>> 1 in d
False
```

[Example 3-7](#) implements a class `StrKeyDict0` that passes the preceding doctests.



## TIP

A better way to create a user-defined mapping type is to subclass `collections.UserDict` instead of `dict` (as we'll do in [Example 3-8](#)). Here we subclass `dict` just to show that `__missing__` is supported by the built-in `dict.__getitem__` method.

*Example 3-7. `StrKeyDict0` converts nonstring keys to `str` on lookup (see tests in [Example 3-6](#))*

```
class StrKeyDict0(dict): ❶

    def __missing__(self, key):
        if isinstance(key, str): ❷
            raise KeyError(key)
        return self[str(key)] ❸

    def get(self, key, default=None):
        try:
            return self[key] ❹
        except KeyError:
            return default ❺

    def __contains__(self, key):
        return key in self.keys() or str(key) in self.keys() ❻
```

- ❶ `StrKeyDict0` inherits from `dict`.
- ❷ Check whether key is already a `str`. If it is, and it's missing, raise `KeyError`.
- ❸ Build `str` from key and look it up.
- ❹ The `get` method delegates to `__getitem__` by using the `self[key]` notation; that gives the opportunity for our `__missing__` to act.
- ❺ If a `KeyError` was raised, `__missing__` already failed, so we return the default.
- ❻ Search for unmodified key (the instance may contain non-`str` keys), then for a `str` built from the key.

Take a moment to consider why the test `isinstance(key, str)` is necessary in the `__missing__` implementation.

Without that test, our `__missing__` method would work OK for any key `k`—`str` or not `str`—whenever `str(k)` produced an existing key. But if `str(k)` is not an existing key, we'd have an infinite recursion. The last line, `self[str(key)]` would call `__getitem__` passing that `str` key, which in turn would call `__missing__` again.

The `__contains__` method is also needed for consistent behavior in this example, because the operation `k in d` calls it, but the method inherited from `dict` does not fall back to invoking `__missing__`. There is a subtle detail in our implementation of

`__contains__`: we do not check for the key in the usual Pythonic way—`k in my_dict`—because `str(key) in self` would recursively call `__contains__`. We avoid this by explicitly looking up the key in `self.keys()`.

#### NOTE

A search like `k in my_dict.keys()` is efficient in Python 3 even for very large mappings because `dict.keys()` returns a view, which is similar to a set, as we'll see in [“Set operations on dict views”](#). However, remember that `k in my_dict` does the same job, and is faster because it avoids the attribute lookup to find the `.keys` method. I had a specific reason to use `self.keys()` in the `__contains__` method in [Example 3-7](#).

The check for the unmodified key—`key in self.keys()`—is necessary for correctness because `StrKeyDict0` does not enforce that all keys in the dictionary must be of type `str`. Our only goal with this simple example is to make searching “friendlier” and not enforce types.

So far we have covered the `dict` and `defaultdict` mapping types, but the standard library comes with other mapping implementations, which we discuss next.

## Variations of dict

In this section, we summarize the various mapping types included in the standard library, besides `defaultdict`, already covered in [“defaultdict: Another Take on Missing Keys”](#).

The following mapping types are ready to instantiate and use:

### *collections.OrderedDict*

Maintains keys in insertion order, allowing iteration over items in a predictable order. The `popitem` method of an `OrderedDict` pops the last item by default, but if called as `my_odict.popitem(last=False)`, it pops the first item added. Now that the built-in `dict` also keeps the keys ordered since Python 3.6, the main reason to use `OrderedDict` is writing code that is backward-compatible with earlier Python versions.

### *collections.ChainMap*

Holds a list of mappings that can be searched as one. The lookup is performed on each mapping in order, and succeeds if the key is found in any of them. This is useful to interpreters for languages with nested scopes, where each mapping represents a scope context. [The “ChainMap objects” section of the `collections`](#)

[docs](#) has several examples of `ChainMap` usage, including this snippet inspired by the basic rules of variable lookup in Python:

```
import builtins
pylookup = ChainMap(locals(), globals(), vars(builtins))
```

### *collections.Counter*

A mapping that holds an integer count for each key. Updating an existing key adds to its count. This can be used to count instances of hashable objects or as a multiset (see below). `Counter` implements the `+` and `-` operators to combine tallies, and other useful methods such as `most_common([n])`, which returns an ordered list of tuples with the *n* most common items and their counts; see the [documentation](#). Here is `Counter` used to count letters in words:

```
>>> ct = collections.Counter('abracadabra')
>>> ct
Counter({'a': 5, 'b': 2, 'r': 2, 'c': 1, 'd': 1})
>>> ct.update('aaaazzz')
>>> ct
Counter({'a': 10, 'z': 3, 'b': 2, 'r': 2, 'c': 1, 'd': 1})
>>> ct.most_common(3)
[('a', 10), ('z', 3), ('b', 2)]
```

Note that the `'b'` and `'r'` keys are tied in third place, but `ct.most_common(3)` shows only three counts.

To use `collections.Counter` as a multiset, each element in the set is a key, and the count is the number of occurrences of that element in the set.

`OrderedDict`, `ChainMap`, and `Counter` are ready to instantiate but can also be customized by subclassing. In contrast, the next mappings are intended as base classes to be extended.

## Building custom mappings

These mapping types are not meant to be instantiated directly, but for subclassing when we need to create custom types:

### *collections.UserDict*

A pure Python implementation of a mapping that behaves like a standard `dict`. See [“Subclassing UserDict”](#) for an extended explanation.

### *typing.TypedDict*

Using new type declaration syntax, the `TypedDict` class added in Python 3.8 lets you create mapping types that only accept a specific set of string keys, with each key constrained to accept only values of a specific type. This is covered in [Link to Come], in [Chapter 5](#).

The `collections.UserDict` class behaves like a `dict`, but it is slower because it is implemented in Python, not in C. We'll cover it in more detail next.

## Subclassing UserDict

It's almost always easier to create a new mapping type by extending `UserDict` rather than `dict`. We realize that when we try to extend our `StrKeyDict0` from [Example 3-7](#) to make sure that any keys added to the mapping are stored as `str`.

The main reason why it's better to subclass `UserDict` rather than `dict` is that the built-in has some implementation shortcuts that end up forcing us to override methods that we can just inherit from `UserDict` with no problems.<sup>5</sup>

Note that `UserDict` does not inherit from `dict`, but uses composition: it has an internal `dict` instance, called `data`, which holds the actual items. This avoids undesired recursion when coding special methods like `__setitem__`, and simplifies the coding of `__contains__`, compared to [Example 3-7](#).

Thanks to `UserDict`, `StrKeyDict` ([Example 3-8](#)) is actually shorter than `StrKeyDict0` ([Example 3-7](#)), but it does more: it stores all keys as `str`, avoiding unpleasant surprises if the instance is built or updated with data containing nonstring keys.

*Example 3-8. `StrKeyDict` always converts non-string keys to `str`—on insertion, update, and lookup*

---

```
import collections

class StrKeyDict(collections.UserDict): ❶

    def __missing__(self, key): ❷
        if isinstance(key, str):
            raise KeyError(key)
        return self[str(key)]

    def __contains__(self, key):
        return str(key) in self.data ❸

    def __setitem__(self, key, item):
        self.data[str(key)] = item ❹
```

❶ `StrKeyDict` extends `UserDict`.

- ② `__missing__` is exactly as in [Example 3-7](#).
- ③ `__contains__` is simpler: we can assume all stored keys are `str` and we can check on `self.data` instead of invoking `self.keys()` as we did in `StrKeyDict0`.
- ④ `__setitem__` converts any key to a `str`. This method is easier to overwrite when we can delegate to the `self.data` attribute.

Because `UserDict` extends `abc.MutableMapping`, the remaining methods that make `StrKeyDict` a full-fledged mapping are inherited from `UserDict`, `MutableMapping`, or `Mapping`. The latter have several useful concrete methods, in spite of being abstract base classes (ABCs). The following methods are worth noting:

#### *`MutableMapping.update`*

This powerful method can be called directly but is also used by `__init__` to load the instance from other mappings, from iterables of `(key, value)` pairs, and keyword arguments. Because it uses `self[key] = value` to add items, it ends up calling our implementation of `__setitem__`.

#### *`Mapping.get`*

In `StrKeyDict0` ([Example 3-7](#)), we had to code our own `get` to obtain results consistent with `__getitem__`, but in [Example 3-8](#) we inherited `Mapping.get`, which is implemented exactly like `StrKeyDict0.get` (see [Python source code](#)).

#### **TIP**

Antoine Pitrou authored [PEP 455 — Adding a key-transforming dictionary to collections](#) and a patch to enhance the `collections` module with a `TransformDict`, that is more general than `StrKeyDict` and preserves the keys as they are provided, before the transformation is applied. PEP 455 was rejected in May, 2015—see [Raymond Hettinger’s rejection message](#). To experiment with `TransformDict`, I extracted Pitrou’s patch from [issue 18986](#) into a standalone module (`03-dict-set/transformdict.py` in the [Fluent Python 2nd edition code repository](#)).

We know there are immutable sequence types, but how about an immutable mapping? Well, there isn’t a real one in the standard library, but a stand-in is available. Read on.

## **Immutable Mappings**

The mapping types provided by the standard library are all mutable, but you may need to guarantee that a user cannot change a mapping by mistake. A concrete use case can be found, again, in a hardware programming library “[The `\_\_missing\_\_` Method](#)”: the `board.pins` mapping represents the physical GPIO pins on the device.

As such, it's nice to prevent inadvertent updates to `board.pins` because the hardware can't be changed via software, so any change in the mapping would make it inconsistent with the physical reality of the device.

Since Python 3.3, the `types` module provides a wrapper class called `MappingProxyType`, which, given a mapping, returns a `mappingproxy` instance that is a read-only but dynamic proxy for the original mapping. This means that updates to the original mapping can be seen in the `mappingproxy`, but changes cannot be made through it. See [Example 3-9](#) for a brief demonstration.

*Example 3-9. MappingProxyType builds a read-only mappingproxy instance from a dict*

```
>>> from types import MappingProxyType
>>> d = {1: 'A'}
>>> d_proxy = MappingProxyType(d)
>>> d_proxy
mappingproxy({1: 'A'})
>>> d_proxy[1] ❶
'A'
>>> d_proxy[2] = 'x' ❷
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'mappingproxy' object does not support item assignment
>>> d[2] = 'B'
>>> d_proxy ❸
mappingproxy({1: 'A', 2: 'B'})
>>> d_proxy[2]
'B'
>>>
```

- ❶ Items in `d` can be seen through `d_proxy`.
- ❷ Changes cannot be made through `d_proxy`.
- ❸ `d_proxy` is dynamic: any change in `d` is reflected.

Here is how this could be used in practice in the hardware programming scenario: the constructor in a concrete `Board` subclass would fill a private mapping with the pin objects, and expose it to clients of the API via a public `.pins` attribute implemented as a `mappingproxy`. That way the clients would not be able to add, remove, or change pins by accident.

Next, we'll cover one of the most powerful features of dictionaries in Python 3: `dict` views.

## Dictionary views

The `dict` instance methods `.keys()`, `.values()`, and `.items()` return instances of classes called `dict_keys`, `dict_values`, and `dict_items`, respectively<sup>6</sup>. These dictionary views are read-only projections of the internal data structures used in the `dict` implementation. They avoid the memory overhead of the equivalent Python 2 methods that returned lists duplicating data already in the target `dict`, and they also replace the old methods that returned iterators.

Example 3-10 shows some basic operations supported by all dictionary views.

*Example 3-10. The `.values()` method returns a view of the values in a `dict`.*

```
>>> d = dict(a=10, b=20, c=30)
>>> values = d.values()
>>> values
dict_values([10, 20, 30]) ❶
>>> len(values) ❷
3
>>> list(values) ❸
[10, 20, 30]
>>> reversed(values) ❹
<dict_reversevalueiterator object at 0x10e9e7310>
>>> values[0] ❺
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'dict_values' object is not subscriptable
```

- ❶ The repr of a view object shows its content.
- ❷ We can query the `len` of a view.
- ❸ Views are iterable, so it's easy to create lists from them.
- ❹ Views implement `__reversed__`, returning a custom iterator.
- ❺ We can't use `[]` to get individual items from a view.

A view object is a dynamic proxy. If the source `dict` is updated, you can immediately see the changes through an existing view. Continuing from Example 3-10:

```
>>> d['z'] = 99
>>> d
{'a': 10, 'b': 20, 'c': 30, 'z': 99}
>>> values
dict_values([10, 20, 30, 99])
```

The classes `dict_keys`, `dict_values`, and `dict_items` are internal: they are not available via `__builtins__` or any standard library module, and even if you get a reference to one of them, you can't use it to create a view from scratch in Python code:

```
>>> values_class = type({}.values())
>>> v = values_class()
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot create 'dict_values' instances
```

The `dict_values` class is the simplest dictionary view—it implements only the `__len__`, `__iter__`, and `__reversed__` special methods. In addition to these methods, `dict_keys` and `dict_items` implement several set methods, almost as many as the `frozenset` class. After we cover sets, we’ll have more to say about `dict_keys` and `dict_items` in “Set operations on dict views”.

Now that we’ve covered most mapping types in the standard library, we’ll review sets.

## Set Theory

Sets are not new in Python, but are still somewhat underused. The `set` type and its immutable sibling `frozenset` first appeared as modules in the Python 2.3 standard library, and were promoted to built-ins in Python 2.6.

### NOTE

In this book, I use the word “set” to refer both to `set` and `frozenset`. When talking specifically about the `set` class, I use constant width font: `set`.

A set is a collection of unique objects. A basic use case is removing duplication:

```
>>> l = ['spam', 'spam', 'eggs', 'spam', 'bacon', 'eggs']
>>> set(l)
{'eggs', 'spam', 'bacon'}
>>> list(set(l))
['eggs', 'spam', 'bacon']
```



### TIP

If you want to remove duplicates but also preserve the order of the first occurrence of each item, you can now use a plain `dict` to do it, like this:

```
>>> dict.fromkeys(l).keys()
dict_keys(['spam', 'eggs', 'bacon'])
>>> list(dict.fromkeys(l).keys())
['spam', 'eggs', 'bacon']
```

Set elements must be hashable. The `set` type is not hashable, so you can't build a `set` with nested `set` instances. But `frozenset` is hashable, so you can have `frozenset` elements inside a `set`.

In addition to enforcing uniqueness, the set types implement many set operations as infix operators, so, given two sets `a` and `b`, `a | b` returns their union, `a & b` computes the intersection, `a - b` the difference, and `a ^ b` the symmetric difference. Smart use of set operations can reduce both the line count and the execution time of Python programs, at the same time making code easier to read and reason about—by removing loops and conditional logic.

For example, imagine you have a large set of email addresses (the `haystack`) and a smaller set of addresses (the `needles`) and you need to count how many `needles` occur in the `haystack`. Thanks to `set` intersection (the `&` operator) you can code that in a simple line (see [Example 3-11](#)).

*Example 3-11. Count occurrences of needles in a haystack, both of type `set`*

```
found = len(needles & haystack)
```

Without the intersection operator, you'd have write [Example 3-12](#) to accomplish the same task as [Example 3-11](#).

*Example 3-12. Count occurrences of needles in a haystack (same end result as [Example 3-11](#))*

```
found = 0
for n in needles:
    if n in haystack:
        found += 1
```

[Example 3-11](#) runs slightly faster than [Example 3-12](#). On the other hand, [Example 3-12](#) works for any iterable objects `needles` and `haystack`, while [Example 3-11](#) requires that both be sets. But, if you don't have sets on hand, you can always build them on the fly, as shown in [Example 3-13](#).

*Example 3-13. Count occurrences of needles in a haystack; these lines work for any iterable types*

```
found = len(set(needles) & set(haystack))

# another way:
found = len(set(needles).intersection(haystack))
```

Of course, there is an extra cost involved in building the sets in [Example 3-13](#), but if either the `needles` or the `haystack` is already a set, the alternatives in [Example 3-13](#) may be cheaper than [Example 3-12](#).

Any one of the preceding examples are capable of searching 1,000 elements in a `haystack` of 10,000,000 items in about 0.3 milliseconds—that’s close to 0.3 microseconds per element.

Besides the extremely fast membership test (thanks to the underlying hash table), the `set` and `frozenset` built-in types provide a rich API to create new sets or, in the case of `set`, to change existing ones. We will discuss the operations shortly, but first a note about syntax.

## Set Literals

The syntax of `set` literals—`{1}`, `{1, 2}`, etc.—looks exactly like the math notation, with one important exception: there’s no literal notation for the empty `set`, so we must remember to write `set()`.

### SYNTAX QUIRK

Don’t forget: to create an empty `set`, you should use the constructor without an argument: `set()`. If you write `{}`, you’re creating an empty `dict`—this hasn’t changed in Python 3.

In Python 3, the standard string representation of sets always uses the `{...}` notation, except for the empty set:

```
>>> s = {1}
>>> type(s)
<class 'set'>
>>> s
{1}
>>> s.pop()
1
>>> s
set()
```

Literal `set` syntax like `{1, 2, 3}` is both faster and more readable than calling the constructor (e.g., `set([1, 2, 3])`). The latter form is slower because, to evaluate it, Python has to look up the `set` name to fetch the constructor, then build a list, and finally pass it to the constructor. In contrast, to process a literal like `{1, 2, 3}`, Python runs a specialized `BUILD_SET` bytecode<sup>7</sup>.

There is no special syntax to represent `frozenset` literals—they must be created by calling the constructor. The standard string representation in Python 3 looks like a `frozenset` constructor call. Note the output in the console session:

```
>>> frozenset(range(10))
frozenset({0, 1, 2, 3, 4, 5, 6, 7, 8, 9})
```

Speaking of syntax, the idea of listcomps was adapted to build sets as well.

## Set Comprehensions

Set comprehensions (*setcomps*) were added in Python 2.7, together with the dictcomps that we saw in “[dict Comprehensions](#)”. [Example 3-14](#) shows how.

*Example 3-14. Build a set of Latin-1 characters that have the word “SIGN” in their Unicode names*

```
>>> from unicodedata import name ❶
>>> {chr(i) for i in range(32, 256) if 'SIGN' in name(chr(i), '')} ❷
{'$', '=', 'ç', '#', '¤', '<', '¥', 'µ', '×', '$', '¶', '£', '©',
'°', '+', '÷', '±', '>', '¬', '®', '%'}
```

- ❶ Import `name` function from `unicodedata` to obtain character names.
- ❷ Build set of characters with codes from 32 to 255 that have the word `'SIGN'` in their names.

Syntax matters aside, let’s now review the rich assortment of operations provided by sets.

## Set Operations

[Figure 3-2](#) gives an overview of the methods you can use on mutable and immutable sets. Many of them are special methods that overload operators such as `&` and `>=`. [Table 3-2](#) shows the math set operators that have corresponding operators or methods in Python. Note that some operators and methods perform in-place changes on the target set (e.g., `&=`, `difference_update`, etc.). Such operations make no sense in the ideal world of mathematical sets, and are not implemented in `frozenset`.

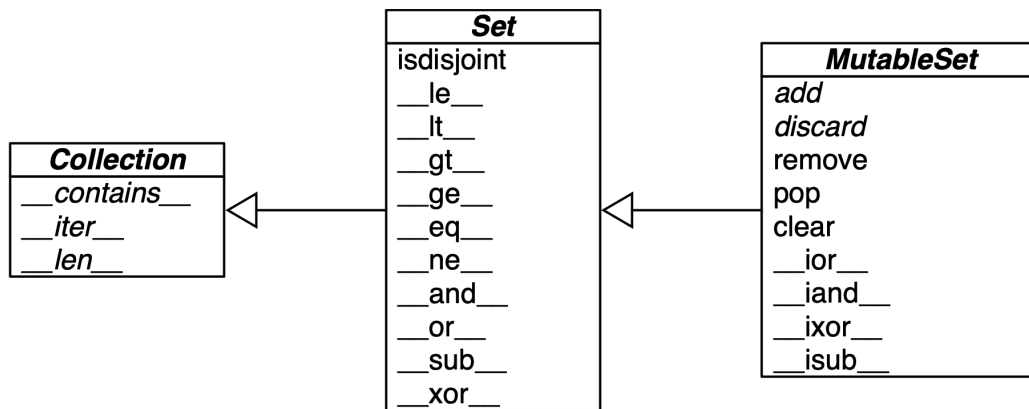


Figure 3-2. Simplified UML class diagram for *MutableSet* and its superclasses from *collections.abc* (names in italic are abstract classes and abstract methods; reverse operator methods omitted for brevity)

### TIP

The infix operators in Table 3-2 require that both operands be sets, but all other methods take one or more iterable arguments. For example, to produce the union of four collections, *a*, *b*, *c*, and *d*, you can call *a.union(b, c, d)*, where *a* must be a set, but *b*, *c*, and *d* can be iterables of any type.

Table 3-2. Mathematical set operations: these methods either produce a new set or update the target set in place, if it's mutable

Math symbol	Python operator	Method	Description
$S \cap Z$	<code>s &amp; z</code>	<code>s.__and__(z)</code>	Intersection of <code>s</code> and <code>z</code>
	<code>z &amp; s</code>	<code>s.__rand__(z)</code>	Reversed <code>&amp;</code> operator
		<code>s.intersection(it, ...)</code>	Intersection of <code>s</code> and all sets built from iterables <code>it</code> , etc.
	<code>s &amp;= z</code>	<code>s.__iand__(z)</code>	<code>s</code> updated with intersection of <code>s</code> and <code>z</code>
		<code>s.intersection_update(it, ...)</code>	<code>s</code> updated with intersection of <code>s</code> and all sets built from iterables <code>it</code> , etc.
$S \cup Z$	<code>s   z</code>	<code>s.__or__(z)</code>	Union of <code>s</code> and <code>z</code>
	<code>z   s</code>	<code>s.__ror__(z)</code>	Reversed <code> </code>
		<code>s.union(it, ...)</code>	Union of <code>s</code> and all sets built from iterables <code>it</code> , etc.
	<code>s  = z</code>	<code>s.__ior__(z)</code>	<code>s</code> updated with union of <code>s</code> and <code>z</code>
		<code>s.update(it, ...)</code>	<code>s</code> updated with union of <code>s</code> and all sets built from iterables <code>it</code> , etc.
$S \setminus Z$	<code>s - z</code>	<code>s.__sub__(z)</code>	Relative complement or difference between <code>s</code> and <code>z</code>
	<code>z - s</code>	<code>s.__rsub__(z)</code>	Reversed <code>-</code> operator
		<code>s.difference(it, ...)</code>	Difference between <code>s</code> and all sets built from iterables <code>it</code> , etc.
	<code>s -= z</code>	<code>s.__isub__(z)</code>	<code>s</code> updated with difference between <code>s</code> and <code>z</code>
		<code>s.difference_update(it, ...)</code>	<code>s</code> updated with difference between <code>s</code> and all sets built from iterables <code>it</code> , etc.
		<code>s.symmetric_difference(it)</code>	Complement of <code>s &amp; set(it)</code>
$S \Delta Z$	<code>s ^ z</code>	<code>s.__xor__(z)</code>	Symmetric difference (the complement of the intersection <code>s &amp; z</code> )
	<code>z ^ s</code>	<code>s.__rxor__(z)</code>	Reversed <code>^</code> operator

Math symbol	Python operator	Method	Description
		<code>s.symmetric_difference_update(it, ...)</code>	s updated with symmetric difference of s and all sets built from iterables it, etc.
$s \wedge= z$		<code>s.__ixor__(z)</code>	s updated with symmetric difference of s and z

Table 3-3 lists set predicates: operators and methods that return True or False.

*Table 3-3. Set comparison operators and methods that return a bool*

Math symbol	Python operator	Method	Description
		<code>s.isdisjoint(z)</code>	s and z are disjoint (no elements in common)
$e \in S$	<code>e in s</code>	<code>s.__contains__(e)</code>	Element e is a member of s
$S \subseteq Z$	<code>s &lt;= z</code>	<code>s.__le__(z)</code>	s is a subset of the z set
		<code>s.issubset(it)</code>	s is a subset of the set built from the iterable it
$S \subset Z$	<code>s &lt; z</code>	<code>s.__lt__(z)</code>	s is a proper subset of the z set
$S \supseteq Z$	<code>s &gt;= z</code>	<code>s.__ge__(z)</code>	s is a superset of the z set
		<code>s.issuperset(it)</code>	s is a superset of the set built from the iterable it
$S \supset Z$	<code>s &gt; z</code>	<code>s.__gt__(z)</code>	s is a proper superset of the z set

In addition to the operators and methods derived from math set theory, the set types implement other methods of practical use, summarized in Table 3-4.

Table 3-4. Additional set methods

	set	frozenset	
<code>s.add(e)</code>	•		Add element <code>e</code> to <code>s</code>
<code>s.clear()</code>	•		Remove all elements of <code>s</code>
<code>s.copy()</code>	•	•	Shallow copy of <code>s</code>
<code>s.discard(e)</code>	•		Remove element <code>e</code> from <code>s</code> if it is present
<code>s.__iter__()</code>	•	•	Get iterator over <code>s</code>
<code>s.__len__()</code>	•	•	<code>len(s)</code>
<code>s.pop()</code>	•		Remove and return an element from <code>s</code> , raising <code>KeyError</code> if <code>s</code> is empty
<code>s.remove(e)</code>	•		Remove element <code>e</code> from <code>s</code> , raising <code>KeyError</code> if <code>e not in s</code>

This completes our overview of the features of sets. As promised in “[Dictionary views](#)”, we’ll now see how two of the dictionary view types behave very much like a `frozenset`.

## Set operations on dict views

Table 3-5 shows that the view objects returned by the `dict` methods `.keys()` and `.items()` are remarkably similar to `frozenset`.

Table 3-5. Methods implemented by *frozenset*, *dict\_keys*, and *dict\_items*.

	<b>frozenset</b>	<b>dict_keys</b>	<b>dict_items</b>	<b>Description</b>
<code>s.__and__(z)</code>	•	•	•	<code>s &amp; z</code> (intersection of <code>s</code> and <code>z</code> )
<code>s.__rand__(z)</code>	•	•	•	Reversed <code>&amp;</code> operator
<code>s.__contains__(e)</code>	•	•	•	<code>e in s</code>
<code>s.copy()</code>	•			Shallow copy of <code>s</code>
<code>s.difference(it, ...)</code>	•			Difference between <code>s</code> and iterables <code>it</code> , etc.
<code>s.intersection(it, ...)</code>	•			Intersection of <code>s</code> and iterables <code>it</code> , etc.
<code>s.isdisjoint(z)</code>	•	•	•	<code>s</code> and <code>z</code> are disjoint (no elements in common)
<code>s.issubset(it)</code>	•			<code>s</code> is a subset of iterable <code>it</code>
<code>s.issuperset(it)</code>	•			<code>s</code> is a superset of iterable <code>it</code>
<code>s.__iter__()</code>	•	•	•	Get iterator over <code>s</code>
<code>s.__len__()</code>	•	•	•	<code>len(s)</code>
<code>s.__or__(z)</code>	•	•	•	<code>s   z</code> (union of <code>s</code> and <code>z</code> )
<code>s.__ror__(z)</code>	•	•	•	Reversed <code> </code> operator
<code>s.__reversed__()</code>		•	•	Get iterator over <code>s</code> in reverse order
<code>s.__rsub__(z)</code>	•	•	•	Reversed <code>-</code> operator
<code>s.__sub__(z)</code>	•	•	•	<code>s - z</code> (difference between <code>s</code> and <code>z</code> )
<code>s.symmetric_difference(it)</code>	•			Complement of <code>s &amp; set(it)</code>
<code>s.union(it, ...)</code>	•			Union of <code>s</code> and iterables <code>it</code> , etc.
<code>s.__xor__(z)</code>	•	•	•	<code>s ^ z</code> (symmetric difference of <code>s</code> and <code>z</code> )
<code>s.__rxor__(z)</code>	•	•	•	Reversed <code>^</code> operator



In particular, `dict_keys` and `dict_items` implement the special methods to support the powerful set operators `&` (intersection), `|` (union), `-` (difference) and `^` (symmetric difference).

This means, for example, that finding the keys that appear in two dictionaries is as easy as this:

```
>>> d1 = dict(a=1, b=2, c=3, d=4)
>>> d2 = dict(b=20, d=40, e=50)
>>> d1.keys() & d2.keys()
{'b', 'd'}
```

Note that the return value of `&` is a `set`. Even better: the set operators in dictionary views are compatible with `set` instances. Check this out:

```
>>> s = {'a', 'e', 'i'}
>>> d1.keys() & s
{'a'}
>>> d1.keys() | s
{'a', 'c', 'b', 'd', 'i', 'e'}
```

This will save a lot of loops and ifs when inspecting the contents of dictionaries in your code.

We now change gears to discuss how sets and dictionaries are implemented with hash tables. After reading the rest of this chapter, you should no longer be surprised by the behavior of `dict`, `set`, and other data structures powered by hash tables.

## Internals of sets and dicts

Understanding how Python dictionaries and sets are built with hash tables is helpful to make sense of their strengths and limitations.

### NOTE

Consider this section optional. You don't need to know all of these details to make good use of dictionaries and sets. But the implementation ideas are beautiful—that's why I covered them. For practical advice, you can skip to [“Practical Consequences of How Sets Work”](#) and [“Practical Consequences of How dict Works”](#).

Here are some questions this section will answer:

- How efficient are Python `dict` and `set`?
- Why are `set` elements unordered?
- Why can't we use any Python object as a `dict` key or `set` element?
- Why does the order of the `dict` keys depend on insertion order?
- Why does the order of `set` elements seem random?

To motivate the study of hash tables, we start by showcasing the amazing performance of `dict` and `set` with a simple test involving millions of items.

## A Performance Experiment

From experience, all Pythonistas know that dicts and sets are fast. We'll confirm that with a controlled experiment.

To see how the size of a `dict`, `set`, or `list` affects the performance of search using the `in` operator, I generated an array of 10 million distinct double-precision floats, the "haystack." I then generated an array of needles: 1,000 floats, with 500 picked from the haystack and 500 verified not to be in it.

For the `dict` benchmark, I used `dict.fromkeys()` to create a `dict` named `haystack` with 1,000 floats. This was the setup for the `dict` test. The actual code I clocked with the `timeit` module is [Example 3-15](#) (like [Example 3-12](#)).

*Example 3-15. Search for needles in haystack and count those found*

```
found = 0
for n in needles:
    if n in haystack:
        found += 1
```

I repeated the benchmark five times, each time increasing tenfold the size of `haystack`, from 1,000 to 10,000,000 items. The result of the `dict` performance test is in [Table 3-6](#).

Table 3-6. Total time for using `in` operator to search for 1,000 needles in haystack dicts of five sizes on a 2.2 GHz Core i7 laptop running Python 3.8.0 (tests timed the loop in Example 3-15)

len of haystack	Factor	dict time	Factor
1,000	1×	0.099ms	1.00×
10,000	10×	0.109ms	1.10×
100,000	100×	0.156ms	1.58×
1,000,000	1,000×	0.372ms	3.76×
10,000,000	10,000×	0.512ms	5.17×

In concrete terms, to check for the presence of 1,000 floating-point keys in a dictionary with 1,000 items, the processing time on my laptop was 99 $\mu$ s, and the same search in a `dict` with 10,000,000 items took 512 $\mu$ s. In other words, the average time for each search in the haystack with 10 million items was 0.512 $\mu$ s—yes, that’s about half microsecond per needle. When the search space became 10,000 times larger, the search time increased a little over 5 times. Nice.

To compare with other collections, I repeated the benchmark with the same haystacks of increasing size, but storing the haystack as a `set` or as `list`. For the `set` tests, in addition to timing the `for` loop in Example 3-15, I also timed the one-liner in Example 3-16, which produces the same result: count the number of elements from `needles` that are also in `haystack`—if both are sets.

Example 3-16. Use set intersection to count the needles that occur in haystack

```
found = len(needles & haystack)
```

Table 3-7 shows the tests side by side. The best times are in the “set& time” column, which displays results for the `set &` operator using the code from Example 3-16. As expected, the worst times are in the “list time” column, because there is no hash table to support searches with the `in` operator on a `list`, so a full scan must be made if the needle is not present, resulting in times that grow linearly with the size of the haystack.

*Table 3-7. Total time for using in operator to search for 1,000 keys in haystacks of 5 sizes, stored as dicts, sets, and lists on a 2.2 GHz Core i7 laptop running Python 3.8.0 (tests timed the loop in Example 3-15 except the set&, which uses Example 3-16)*

len of haystack	Factor	dict time	Factor	set time	Factor	set& time	Factor	list time	Factor
1,000	1×	0.099ms	1.00×	0.107ms	1.00×	0.083ms	1.00×	9.115ms	1.00×
10,000	10×	0.109ms	1.10×	0.119ms	1.11×	0.094ms	1.13×	78.219ms	8.58×
100,000	100×	0.156ms	1.58×	0.147ms	1.37×	0.122ms	1.47×	767.975ms	84.25×
1,000,000	1,000×	0.372ms	3.76×	0.264ms	2.47×	0.240ms	2.89×	8,020.312ms	879.90×
10,000,000	10,000×	0.512ms	5.17×	0.330ms	3.08×	0.298ms	3.59×	78,558.771ms	8,618.63×

If your program does any kind of I/O, the lookup time for keys in dicts or sets is negligible, regardless of the dict or set size (as long as it fits in RAM). See the code used to generate Table 3-7 and accompanying discussion in [Link to Come], [Link to Come].

Now that we have concrete evidence of the speed of dicts and sets, let's explore how that is achieved with the help of hash tables.

## Set hash tables under the hood

Hash tables are a wonderful invention. Let's see how a hash table is used when adding elements to a set.

Let's say we have a set with abbreviated workdays, created like this:

```
>>> workdays = {'Mon', 'Tue', 'Wed', 'Thu', 'Fri'}
>>> workdays
{'Tue', 'Mon', 'Wed', 'Fri', 'Thu'}
```

The core data structure of a Python set is a hash table with at least 8 rows. Traditionally, the rows in hash table are called *buckets*<sup>8</sup>.

A hash table holding the elements of workdays looks like Figure 3-3.

	hash code	pointer to element
0	-1	
1	-1	
2	2414279730484651250	—————→ 'Tue'
3	4199492796428269555	—————→ 'Mon'
4	-5145319347887138165	—————→ 'Wed'
5	7021641685991143771	—————→ 'Fri'
6	-1	
7	-1139383146578602409	—————→ 'Thu'

Figure 3-3. Hash table for the set {'Mon', 'Tue', 'Wed', 'Thu', 'Fri'}. Each bucket has two fields: the hash code and a pointer to the element value. Empty buckets have -1 in the hash code field. The ordering looks random.

In CPython built for a 64-bit CPU, each bucket in a set has two fields: a 64-bit hash code, and a 64-bit pointer to the element value—which is a Python object stored elsewhere in memory. Because buckets have a fixed size, access to an individual bucket is done by offset. There is no field for the indexes from 0 to 7 in Figure 3-3.

Before covering the hash table algorithm, we need to know more about hash codes, and how they relate to equality.

### HASHES AND EQUALITY

The `hash()` built-in function works directly with built-in types and falls back to calling `__hash__` for user-defined types. If two objects compare equal, their hash codes must also be equal, otherwise the hash table algorithm does not work. For example, because `1 == 1.0` is `True`, `hash(1) == hash(1.0)` must also be `True`, even though the internal representation of an `int` and a `float` are very different.<sup>9</sup>

Also, to be effective as hash table indexes, hash codes should scatter around the index space as much as possible. This means that, ideally, objects that are similar but not equal should have hash codes that differ widely. Example 3-17 is the output of a script to compare the bit patterns of hash codes. Note how the hashes of 1 and 1.0 are the same, but those of 1.0001, 1.0002, and 1.0003 are very different.

Example 3-17. Comparing hash bit patterns of 1, 1.0001, 1.0002, and 1.0003 on a 32-bit build of Python (bits that are different in the hashes above and below are highlighted with ! and the right column shows the number of bits that differ)

```
32-bit Python build
1      00000000000000000000000000000001
                                     != 0
```

```

1.0      00000000000000000000000000000001
-----
1.0      00000000000000000000000000000001
        ! !!! ! !! ! !      ! ! !! !!!   != 16
1.0001   00101110101101010000101011011101
-----
1.0001   00101110101101010000101011011101
        !!!  !!!!  !!!!!   !!!!! !!  !   != 20
1.0002   01011101011010100001010110111001
-----
1.0002   01011101011010100001010110111001
        ! !   ! !!! ! !   !! ! !   ! !!!!! != 17
1.0003   000011000001111110010000010010110
-----

```

The code to produce [Example 3-17](#) is in [\[Link to Come\]](#). Most of it deals with formatting the output, but it is listed as [\[Link to Come\]](#) for completeness.

### NOTE

Starting with Python 3.3, a random salt value is included when computing hash codes for `str`, `bytes`, and `datetime` objects, as documented in [Issue 13703—Hash collision security issue](#). The salt value is constant within a Python process but varies between interpreter runs. With PEP-456, Python 3.4 adopted the SipHash cryptographic function to compute hash codes for `str` and `bytes` objects. The random salt and SipHash are security measures to prevent DoS attacks. Details are in a note in the documentation for the `__hash__` special method.

## HASH COLLISIONS

As mentioned, on 64-bit CPython a hash code is a 64-bit number, and that's  $2^{64}$  possible values—which is more than  $10^{19}$ . But most Python types can represent many more different values. For example, a string made of 10 ASCII printable characters picked at random has  $100^{10}$  possible values—more than  $2^{66}$ . Therefore, the hash code of an object usually has less information than the actual object value. This means that objects that are different may have the same hash code.

### TIP

When correctly implemented, hashing guarantees that different hash codes always imply different objects, but the reverse is not true: different objects don't always have different hash codes. When different objects have the same hash code, that's a *hash collision*.

With this basic understanding of hash codes and object equality, we are ready to dive into the algorithm that makes hash tables work, and how hash collisions are handled.

## The hash table algorithm

We will focus on the internals of `set` first, and later transfer the concepts to `dict`.

### NOTE

This is a simplified view of how Python uses a hash table to implement a `set`. For all details, see commented source code for CPython’s `set` and `frozenset` in `Include/setobject.h` and `Objects/setobject.c`.

Let’s see how Python builds a set like `{'Mon', 'Tue', 'Wed', 'Thu', 'Fri'}`, step by step. The algorithm is illustrated by the flowchart in [Figure 3-4](#), and described next.

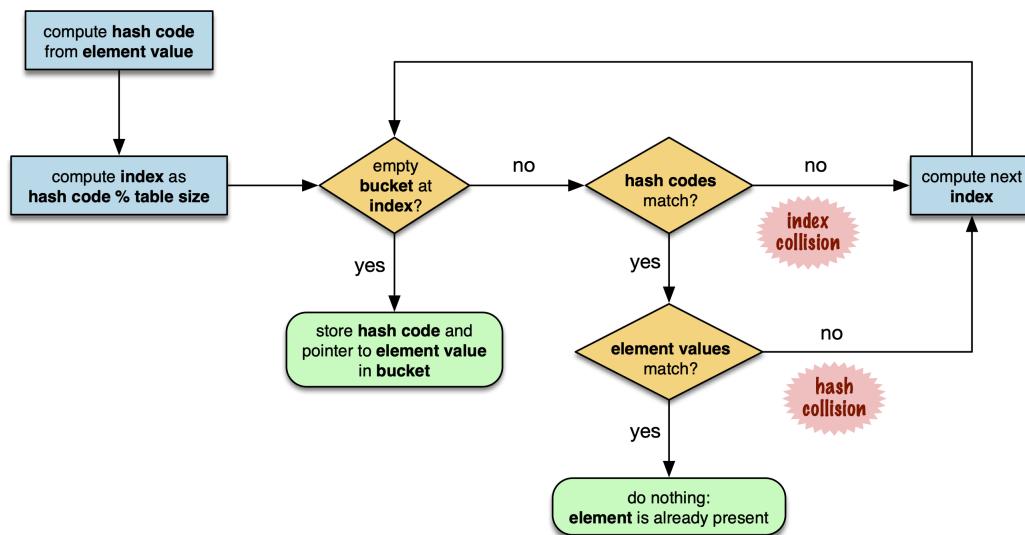


Figure 3-4. Flowchart for algorithm to add element to the hash table of a set.

### Step 0: initialize hash table

As mentioned earlier, the hash table for a `set` starts with 8 empty buckets. As elements are added, Python makes sure at least  $\frac{1}{3}$  of the buckets are empty—doubling the size of the hash table when more space is needed. The hash code field of each bucket is initialized with -1, which means “no hash code”<sup>10</sup>.

### Step 1: compute the hash code for the element

Given the literal `{'Mon', 'Tue', 'Wed', 'Thu', 'Fri'}`, Python gets the hash code for the first element, `'Mon'`. For example, here is a realistic hash code for

'Mon' —you'll probably get a different result because of the random salt Python uses to compute the hash code of strings:

```
>>> hash('Mon')
4199492796428269555
```

### Step 2: probe hash table at index derived from hash code

Python takes the modulus of the hash code with the table size to find a hash table index. Here the table size is 8, and the modulus is 3:

```
>>> 4199492796428269555 % 8
3
```

Probing consists of computing the index from the hash, then looking at the corresponding bucket in the hash table. In this case, Python looks at the bucket at offset 3 and finds -1 in the hash code field, marking an empty bucket.

### Step 3: put the element in the empty bucket

Python stores the hash code of the new element, 4199492796428269555, in the hash code field at offset 3, and a pointer to the string object 'Mon' in the element field. Figure 3-5 shows the current state of the hash table.

	hash code	pointer to element
0	-1	
1	-1	
2	-1	
3	4199492796428269555	→ 'Mon'
4	-1	
5	-1	
6	-1	
7	-1	

Figure 3-5. Hash table for the set {'Mon'}.

## STEPS FOR REMAINING ITEMS

For the second element, 'Tue', steps 1, 2, 3 above are repeated. The hash code for 'Tue' is 2414279730484651250, and the resulting index is 2.



```
>>> hash('Tue')
2414279730484651250
>>> hash('Tue') % 8
2
```

The hash and pointer to element 'Tue' are placed in bucket 2, which was also empty. Now we have Figure 3-6

	hash code	pointer to element
0	-1	
1	-1	
2	2414279730484651250	→ 'Tue'
3	4199492796428269555	→ 'Mon'
4	-1	
5	-1	
6	-1	
7	-1	

Figure 3-6. Hash table for the set {'Mon', 'Tue'}. Note that element ordering is not preserved in the hash table.

## STEPS FOR A COLLISION

When adding 'Wed' to the set, Python computes the hash -5145319347887138165 and index 3. Python probes bucket 3 and sees that it is already taken. But the hash code stored there, 4199492796428269555 is different. As discussed in “[Hashes and equality](#)”, if two objects have different hashes, then their value is also different. This is an index collision. Python then probes the next bucket and finds it empty. So 'Wed' ends up at index 4, as shown in [Figure 3-7](#).

	hash code	pointer to element
0	-1	
1	-1	
2	2414279730484651250	—————→ 'Tue '
3	4199492796428269555	—————→ 'Mon '
4	-5145319347887138165	—————→ 'Wed '
5	-1	
6	-1	
7	-1	

Figure 3-7. Hash table for the set { 'Mon', 'Tue', 'Wed' }. After the collision, 'Wed' is put at index 4.

Adding the next element, 'Thu', is boring: there's no collision, and it lands in its natural bucket, at index 7.

Placing 'Fri' is more interesting. Its hash, 7021641685991143771 implies index 3, which is taken by 'Mon'. Probing the next bucket—4—Python finds the hash for 'Wed' stored there. The hash codes don't match, so this is another index collision. Python probes the next bucket. It's empty, so 'Fri' ends up at index 5. The end state of the hash table is shown in [Figure 3-8](#).

#### NOTE

Incrementing the index after a collision is called *linear probing*. This can lead to clusters of occupied buckets, which can degrade the hash table performance, so CPython counts the number of linear probes and after a certain threshold, applies a pseudo random number generator to obtain a different index from other bits of the hash code. This optimization is particularly important in large sets.

	hash code	pointer to element
0	-1	
1	-1	
2	2414279730484651250	—————→ 'Tue '
3	4199492796428269555	—————→ 'Mon '
4	-5145319347887138165	—————→ 'Wed '
5	7021641685991143771	—————→ 'Fri '
6	-1	
7	-1139383146578602409	—————→ 'Thu '

Figure 3-8. Hash table for the set {'Mon', 'Tue', 'Wed', 'Thu', 'Fri'}. It is now 62.5% full—close to the  $\frac{2}{3}$  threshold.

When there is an element in the probed bucket and the hash codes match, Python also needs to compare the actual object values. That’s because, as explained in “[Hash collisions](#)”, it’s possible that two different objects have the same hash code—although that’s rare for strings, thanks to the quality of the Siphash algorithm<sup>11</sup>. This explains why hashable objects must implement both `__hash__` and `__eq__`.

If a new element were added to our example hash table, it would be more than  $\frac{2}{3}$  full, therefore increasing the chances of index collisions. To prevent that, Python would allocate a new hash table with 16 buckets, and reinsert all elements there.

All this may seem like a lot of work, but even with millions of items in a `set`, many insertions happen with no collisions, and the average number of collisions per insertion is between one and two. Under normal usage, even the unluckiest elements can be placed after a handful of collisions are resolved.

Now, given what we’ve seen so far, follow the flowchart in [Figure 3-4](#) to answer the following puzzle without using the computer.

Given the following `set`, what happens when you add an integer 1 to it?

```
>>> s = {1.0, 2.0, 3.0}
>>> s.add(1)
```

How many elements are in `s` now? Does 1 replace the element 1.0? When you have your answer, use the Python console to verify it.

## SEARCHING ELEMENTS IN A HASH TABLE

Consider the `workdays` set with the hash table shown in [Figure 3-8](#). Is `'Sat'` in it? This is the simplest execution path for the expression `'Sat' in workdays`:

1. Call `hash('Sat')` to get a hash code. Let's say it is 4910012646790914166
2. Derive a hash table index from the hash code, using `hash_code % table_size`. In this case, the index is 6.
3. Probe offset 6: it's empty. This means `'Sat'` is not in the set. Return `False`.

Now consider the simplest path for an element that is present in the set. To evaluate `'Thu' in workdays`:

1. Call `hash('Tue')`. Pretend result is 6166047609348267525.
2. Compute index: `6166047609348267525 % 8` is 5.
3. Probe offset 5:
  - a. Compare hash codes. They are equal.
  - b. Compare the object values. They are equal. Return `True`.

Collisions are handled in the way described when adding an element. In fact, the flowchart in [Figure 3-4](#) applies to searches as well, with the exception of the terminal nodes—the rectangles with rounded corners. If an empty bucket is found, the element is not present, so Python returns `False`; otherwise, when both the hash code and the values of the sought element match an element in the hash table, the return is `True`.

## PRACTICAL CONSEQUENCES OF HOW SETS WORK

The `set` and `frozenset` types are both implemented with a hash table, which has these effects:

- Set elements must be hashable objects. They must implement proper `__hash__` and `__eq__` methods as described in [“What Is Hashable?”](#).
- Membership testing is very efficient. A set may have millions of elements, but the bucket for an element can be located directly by computing the hash code of the element and deriving an index offset, with the possible overhead of a small number of probes to find a matching element or an empty bucket.
- Sets have a significant memory overhead. The most compact internal data structure for a container would be an array of pointers<sup>12</sup>. Compared to that, a hash table adds a hash code per entry, and at least  $\frac{1}{3}$  of empty buckets to minimize collisions.
- Element ordering depends on insertion order, but not in a useful or reliable way. If two elements are involved in a collision, the bucket where each is stored depends on which element is added first.

- Adding elements to a set may change the order of other elements. That’s because, as the hash table is filled, Python may need to recreate it to keep at least  $\frac{1}{3}$  of the buckets empty. When this happens, elements are reinserted and different collisions may occur.

## Hash table usage in dict

Since 2012, the implementation of the `dict` type had two major optimizations to reduce memory usage. The first one was proposed as [PEP 412 — Key-Sharing Dictionary](#) and implemented in Python 3.3<sup>13</sup>. The second is called “compact dict”, and landed in Python 3.6. As a side effect, the compact dict space optimization preserves key insertion order. In the next sections we’ll discuss the compact dict and the new key-sharing scheme—in this order, for easier presentation.

## HOW COMPACT DICT SAVES SPACE

### NOTE

This is a high level explanation of the Python dict implementation. One difference is that the actual usable fraction of a dict hash table is  $\frac{1}{3}$ , and not  $\frac{2}{3}$  as in sets. The actual  $\frac{1}{3}$  fraction would require 16 buckets to hold the 4 items in my example dict, and the diagrams in this section would become too tall, so I pretend the usable fraction is  $\frac{2}{3}$  in these explanations. One comment in `Objects/dictobject.c` explains that any fraction between  $\frac{1}{3}$  and  $\frac{2}{3}$  “seem to work well in practice”.

Consider a dict holding the abbreviated names for the weekdays from 'Mon' through 'Thu', and the number of students enrolled in swimming class on each day:

```
>>> swimmers = {'Mon': 14, 'Tue': 12, 'Wed': 14, 'Thu': 11}
```

Before the compact dict optimization, the hash table underlying the `swimmers` dictionary would look like [Figure 3-9](#). As you can see, in a 64-bit Python, each bucket holds three 64-bit fields: the hash code of the key, a pointer to the key object, and a pointer to the value object. That’s 24 bytes per bucket.

Old dict hash table			
	hash code	pointer to key object	pointer to value object
0	-1		
1	-1		
2	2414279730484651250	→ 'Tue '	→ 12
3	4199492796428269555	→ 'Mon '	→ 14
4	-5145319347887138165	→ 'Wed '	→ 14
5	-1		
6	-1		
7	-1139383146578602409	→ 'Thu '	→ 11

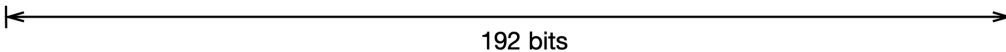

  
192 bits

Figure 3-9. Old hash table format for a `dict` with 4 key-value pairs. Each bucket is a struct with the hash code of the key, a pointer to the key, and a pointer to the value.

The first two fields play the same role as they do in the implementation of sets. To find a key, Python computes the hash code of the key, derives an index from the key, then probes the hash table to find a bucket with a matching hash code and a matching key object. The third field provides the main feature of a `dict`: mapping a key to an arbitrary value. The key must be a hashable object, and the hash table algorithm ensures it will be unique in the `dict`. But the value may be any object—it doesn't need to be hashable or unique.

Raymond Hettinger observed that significant savings could be made if the hash code and pointers to key and value were held in an `entries` array with no empty rows, and the actual hash table were a sparse array with much smaller buckets holding indexes into the `entries` array<sup>14</sup>. In his original [message to `python-dev`](#), Hettinger called the hash table `indices`. The width of the buckets in `indices` varies as the `dict` grows, starting at 8-bits per bucket—enough to index up to 128 entries, while reserving negative values for special purposes, such as -1 for empty and -2 for deleted.

As an example, the `swimmers` dictionary would then be stored as shown in [Figure 3-10](#).

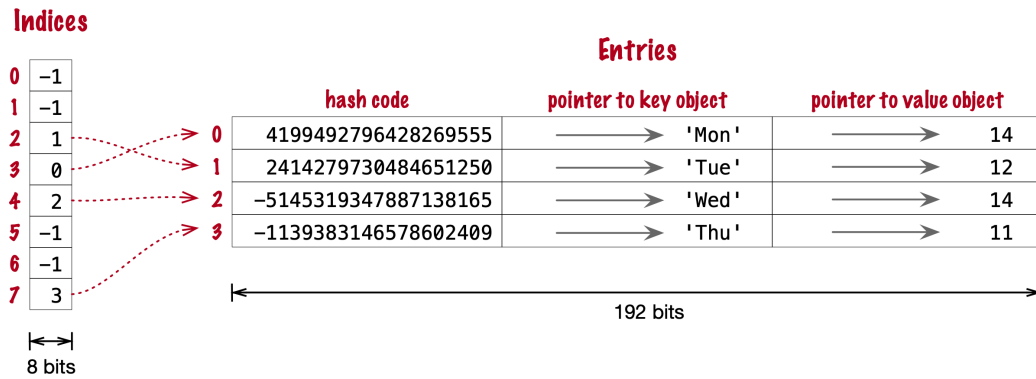


Figure 3-10. Compact storage for a dict with 4 key-value pairs. Hash codes and pointers to keys and values are stored in insertion order in the `entries` array, and the entry offsets derived from the hash codes are held in the `indices` sparse array, where an index value of -1 signals an empty bucket.

Assuming a 64-bit build of CPython, our 4-item swimmers dictionary would take 192 bytes of memory in the old scheme: 24 bytes per bucket, times 8 rows. The equivalent compact dict uses 104 bytes in total: 96 bytes in `entries` (24 \* 4), plus 8 bytes for the buckets in `indices`—configured as an array of 8 bytes.

The next section describes how those two arrays are used.

## ALGORITHM FOR ADDING ITEMS TO COMPACT DICT.

### Step 0: set up indices

The `indices` table is initially set up as an array of signed bytes, with 8 buckets, each initialized with -1 to signal “empty bucket”. Up to 5 of these buckets will eventually hold indices to rows in the `entries` array, leaving 1/3 of them with -1. The other array, `entries`, will hold key/value data with the same three fields as in the old scheme—but in insertion order.

### Step 1: compute hash code for the key

To add the key-value pair ('Mon', 14) to the `swimmers` dictionary, Python first calls `hash('Mon')` to compute the hash code of that key.

### Step 2: probe entries via indices

Python computes `hash('Mon') % len(indices)`. In our example, this is 3. Offset 3 in `indices` holds -1: it’s an empty bucket.

### Step 3: put key-value in entries, updating indices.

The `entries` array is empty, so the next available offset there is 0. Python puts 0 at offset 3 in `indices` and stores the hash code of the key, a pointer to the key object 'Mon', and a pointer to the `int` value 14 at offset 0 in `entries`. Figure 3-11 shows the state of the arrays when the value of `swimmers` is {'Mon': 14}.

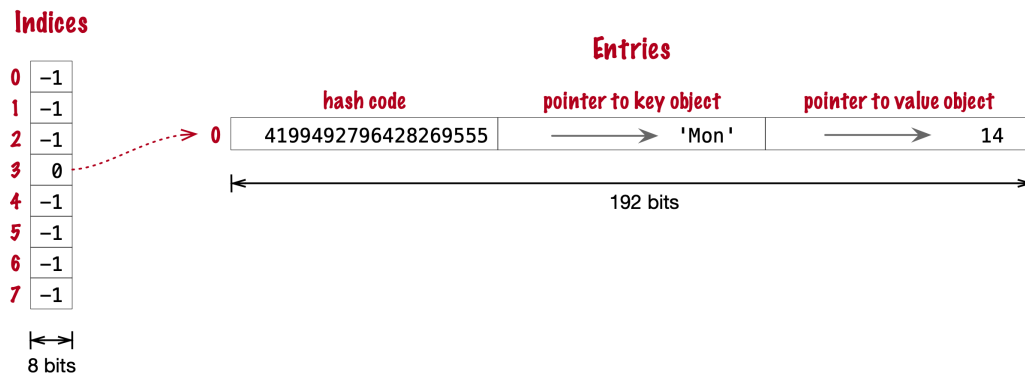


Figure 3-11. Compact storage for the `{'Mon': 14}`: `indices[3]` holds the offset of the first entry: `entries[0]`.

## STEPS FOR NEXT ITEM

To add `('Tue', 12)` to swimmers:

1. Compute hash code of key `'Tue'`.
2. Compute offset into `indices`, as `hash('Tue') % len(indices)`. This is 2. `indices[2]` has -1. No collision so far.
3. Put the next available `entries` offset, 1, in `indices[2]`, then store entry at `entries[1]`.

Now the state is Figure 3-12. Note that `entries` holds the key-value pairs in insertion order.

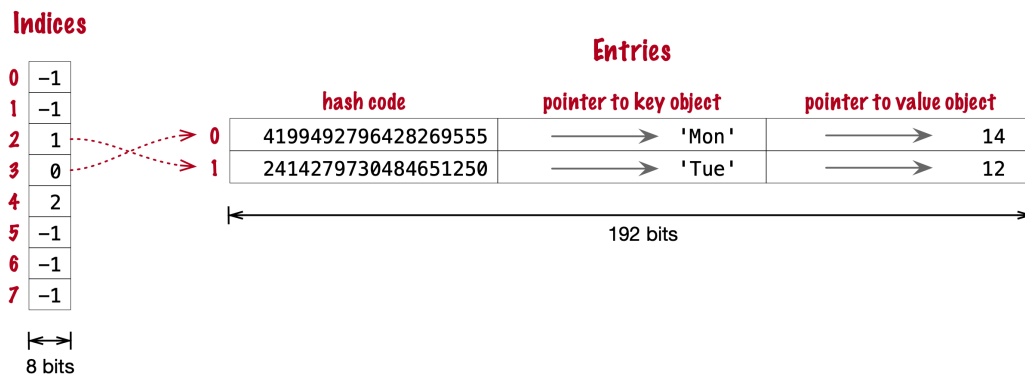


Figure 3-12. Compact storage for the `{'Mon': 14, 'Tue': 12}`.

## STEPS FOR A COLLISION

1. Compute hash code of key `'Wed'`.
2. Now, `hash('Wed') % len(indices)` is 3. `indices[3]` has 0, pointing to an existing entry. Look at the hash code in `entries[0]`. That's the hash code for `'Mon'`, which happens to be different than the hash code for `'Wed'`. This



mismatch signals a collision. Probes the next index: `indices[4]`. That's -1, so it can be used.

3. Make `indices[4] = 2`, because 2 is the next available offset at `entries`. Then fill `entries[2]` as usual.

After adding ('Wed', 14), we have [Figure 3-13](#)

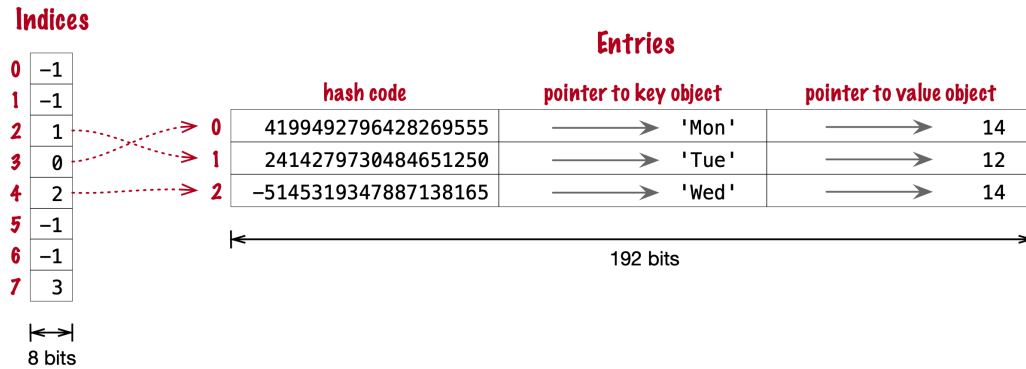


Figure 3-13. Compact storage for the {'Mon': 14, 'Tue': 12, 'Wed': 14}.

## HOW A COMPACT DICT GROWS

Recall that the buckets in the `indices` array are 8 signed bytes initially, enough to hold offsets for up to 5 entries, leaving  $\frac{1}{3}$  of the buckets empty. When the 6th item is added to the `dict`, `indices` is reallocated to 16 buckets—enough for 10 entry offsets. The size of `indices` is doubled as needed, while still holding signed bytes, until the time comes to add the 129th item to the `dict`. At this point, the `indices` array has 256 8-bit buckets. However, a signed byte is not enough to hold offsets after 128 entries, so the `indices` array is rebuilt to hold 256 16-bit buckets to hold signed integers—wide enough to represent offsets to 32,768 rows in the `entries` table. The next resizing happens at the 171st addition, when `indices` would become more than  $\frac{2}{3}$  full. Then the number of buckets in `indices` is doubled to 512, but each bucket still 16-bits wide each. In summary, the `indices` array grows by doubling the number of buckets, and also—less often—by doubling the width of each bucket to accommodate a growing number of rows in `entries`.

This concludes our summary of the compact `dict` implementation. I omitted many details, but now let's take a look at the other space-saving optimization for dictionaries: key-sharing.

## Key-sharing dictionary

Instances of user-defined classes usually hold their attributes in a `__dict__` attribute which is a regular dictionary<sup>15</sup>. In an instance `__dict__`, the keys are the attribute names, and the values are the attribute values. Most of the time, all instances have the same attributes with different values. When that happens, 2 of the 3 fields in the `entries` table for every instance has the exact same content: the hash code of the

attribute name, and a pointer to the attribute name. Only the pointer to the attribute value is different.

In [PEP 412 — Key-Sharing Dictionary](#), Mark Shannon proposed to split the storage of dictionaries used as instance `__dict__`, so that each attribute hash code and pointer is stored only once, linked to the class, and the attribute values are kept in parallel arrays of pointers attached to each instance.

Given a `Movie` class where all instances have the same attributes named `'title'`, `'release'`, `'directors'`, and `'actors'`, [Figure 3-14](#) shows the arrangement of key-sharing in a split dictionary—also implemented with the new compact layout.

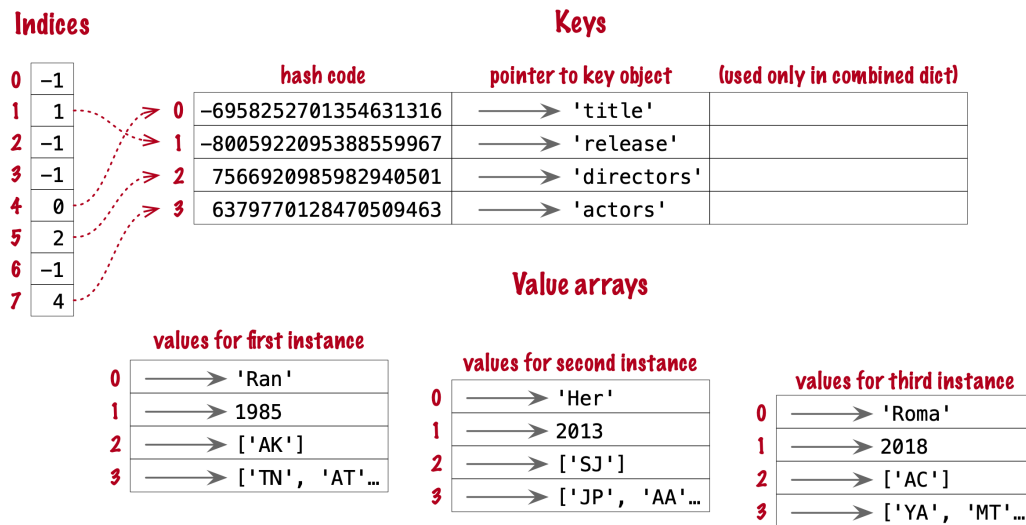


Figure 3-14. Split storage for the `__dict__` of a class and three instances.

PEP 412 introduced the terms *combined-table* to discuss the old layout and *split-table* for the proposed optimization.

The combined-table layout is still the default when you create a `dict` using literal syntax or call `dict()`. A split-table dictionary is created to fill the `__dict__` special attribute of an instance, when it is the first instance of a class. The keys table (see [Figure 3-14](#)) is then cached in the class object. This leverages the fact that most Object Oriented Python code assigns all instance attributes in the `__init__` method. That first instance (and all instances after it) will hold only its own a value array. If an instance gets a new attribute not found in the shared keys table, then this instance's `__dict__` is converted to combined-table form. However, if this instance is the only one in its class, the `__dict__` is converted back to split-table, since it is assumed that further instances will have the same set of attributes and key sharing will be useful.

The `PyDictObject` struct that represents a `dict` in the CPython source code is the same for both *combined-table* and *split-table* dictionaries. When a `dict` converts from one layout to the other, the change happens in `PyDictObject` fields, with the help of other internal data structures.

## Practical Consequences of How dict Works

- Keys must be hashable objects. They must implement proper `__hash__` and `__eq__` methods as described in [“What Is Hashable?”](#).
- Key searches are nearly as fast as element searches in sets.
- Item ordering is preserved in the `entries` table—this was implemented in CPython 3.6, and became an official language feature in 3.7.
- To save memory, avoid creating instance attributes outside of the `__init__` method. If all instance attributes are created in `__init__`, the `__dict__` of your instances will use the split-table layout, sharing the same indices and key entries array stored with the class.

## Chapter Summary

Dictionaries are a keystone of Python. Beyond the basic `dict`, the standard library offers handy, ready-to-use specialized mappings like `defaultdict`, `ChainMap`, and `Counter`, all defined in the `collections` module. With the new `dict` implementation, `OrderedDict` is not as useful as before, but should remain in the standard library for backward-compatibility—and it offers the `.popitem(last=False)` method option to drop and return the first item, which `dict` doesn't yet have. Also in the `collections` module is the easy-to-extend `UserDict` class.

Two powerful methods available in most mappings are `setdefault` and `update`. The `setdefault` method is used to update items holding mutable values, for example, in a `dict` of `list` values, to avoid redundant searches for the same key. The `update` method allows bulk insertion or overwriting of items from any other mapping, from iterables providing `(key, value)` pairs and from keyword arguments. Mapping constructors also use `update` internally, allowing instances to be initialized from mappings, iterables, or keyword arguments.

A clever hook in the mapping API is the `__missing__` method, which lets you customize what happens when a key is not found when using the `d[k]` syntax which invokes `__getitem__`.

The `collections.abc` module provides the `Mapping` and `MutableMapping` abstract base classes as standard interfaces, useful for run-time type checking. The little-known `MappingProxyType` from the `types` module creates immutable mappings. There are also ABCs for `Set` and `MutableSet`.

Dictionary views are a great addition in Python 3, without the memory overhead of the Python 2 `.keys()`, `.values()` and `.items()` methods that built lists duplicating data in the target `dict` instance. In addition, the `dict_keys` and `dict_items` classes support the most useful methods and operators of `frozenset`.

The hash table implementation underlying `set` is extremely fast. Understanding its logic explains why elements are apparently unordered and may even be reordered behind our backs. There is a price to pay for all this speed, and the price is in memory. Finally, we saw how optimizations in the hash tables underlying `dict` save memory and preserve key insertion order.

## Further Reading

In The Python Standard Library documentation, [8.3. collections — Container datatypes](#) includes examples and practical recipes with several mapping types. The Python source code for the module `Lib/collections/__init__.py` is a great reference for anyone who wants to create a new mapping type or grok the logic of the existing

ones. Chapter 1 of *Python Cookbook, Third edition* (O'Reilly) by David Beazley and Brian K. Jones has 20 handy and insightful recipes with data structures—the majority using `dict` in clever ways.

Greg Ganderberger advocates for the continued use of `collections.OrderedDict`, on the grounds that “explicit is better than implicit”, backward compatibility, and the fact that some tools and libraries assume the ordering of `dict` keys is irrelevant—his post: [Python Dictionaries Are Now Ordered. Keep Using OrderedDict.](#)

PEP 3106 — Revamping `dict.keys()`, `.values()` and `.items()` is where Guido van Rossum presented the dictionary views feature for Python 3. In the abstract, he wrote the idea came the Java Collections Framework.

PyPy was the first Python interpreter to implement Raymond Hettinger’s proposal of compact dicts, and they blogged about it in [Faster, more memory efficient and more ordered dictionaries on PyPy](#), acknowledging that a similar layout was adopted in PHP 7, described in [PHP’s new hashtable implementation](#). It’s always great when creators cite prior art.

At PyCon 2017, Brandon Rhodes presented [The Dictionary Even Mightier](#), a sequel to his classic animated presentation [The Mighty Dictionary](#)—including animated hash collisions! Another up-to-date, but more in-depth video on the internals of Python’s `dict` is [Modern Dictionaries](#) by Raymond Hettinger, where he tells that after initially failing to sell compact dicts to the CPython core devs, he lobbied the PyPy team, they adopted it, the idea gained traction, and was finally contributed to CPython 3.6 by INADA Naoki. For all details, check out the extensive comments in the CPython code for [Objects/dictobject.c](#) and [Objects/dict-common.h](#), as well as the design document [Objects/dictnotes.txt](#).

The rationale for adding sets to Python is documented in [PEP 218 — Adding a Built-In Set Object Type](#). When PEP 218 was approved, no special literal syntax was adopted for sets. The `set` literals were created for Python 3 and backported to Python 2.7, along with `dict` and `set` comprehensions. At PyCon 2019, I presented [Set Practice: learning from Python’s set types \(slides\)](#), describing use cases of sets in real programs, covering their API design, and the implementation of `uintset`, a set class for integer elements using a bit vector instead of a hash table, inspired by an example in chapter 6 of the excellent *The Go Programming Language*, by Donovan & Kernighan.

IEEE’s Spectrum magazine has a story about Hans Peter Luhn, a prolific inventor who patented a punched card deck to select cocktail recipes depending on ingredients available, among other diverse inventions including... hash tables! See in [Hans Peter Luhn and the Birth of the Hashing Algorithm](#).

---

## SOAPBOX

### The fundamental theorem of software engineering

My friend Geraldo Cohen once remarked that Python is “simple and correct.”

The following quote is called—with some irony—the *fundamental theorem of software engineering*:

*Any problem in computer science can be solved with another level of indirection<sup>16</sup>.*

—David Wheeler

In the business of software, a useful variation is:

*Any problem in software engineering can be solved with another level of abstraction, except for the problem of too many levels of abstraction.*

—anonymous

Python’s compact dict and the key-sharing scheme are both prime examples of indirection solving problems. But they are transparent to the end user, so they don’t force us to learn new abstractions—they just make our life easier.

Simple and correct.

### Syntactic sugar

Here is another famous computer science quote:

*Syntactic sugar causes cancer of the semicolon.*

—Alan Perlis

Some computer language purists like to dismiss syntax as unimportant, but coders know it matters in practice.

Before finding Python, I had done web programming using Perl, PHP, and JavaScript. The syntax for mappings in these languages is very useful, and I badly miss it whenever I have to use Java or C. A good literal syntax for mappings makes it easy to do configuration, table-driven implementations, and to hold data for prototyping and testing. The lack of it pushed the Java community to adopt the verbose and overly complex XML as a data format.

JSON was proposed as “The Fat-Free Alternative to XML” and became a huge success, replacing XML in many contexts. A concise syntax for lists and dictionaries makes an excellent data interchange format.

PHP and Ruby imitated the hash syntax from Perl, using => to link keys to values. JavaScript followed the lead of Python and uses :. Why use two characters when one is readable enough?

JSON came from JavaScript, but it also happens to be an almost exact subset of Python syntax. JSON is compatible with Python except for the spelling of the values true, false, and null.

Armin Ronacher [tweeted](#) that he likes to hack Python’s \_\_builtin\_\_ to add JSON-compatible aliases for Python’s True, False, and None so he can paste JSON directly in the console. The basic idea:

```
>>> true, false, null = True, False, None
>>> fruit = {
...     "type": "banana",
...     "avg_weight": 123.2,
...     "edible_peel": false,
...     "species": ["acuminata", "balbisiana", "paradisiaca"],
```

```
...     "issues": null,
... }
>>> fruit
{'type': 'banana', 'avg_weight': 123.2, 'edible_peel': False,
'species': ['acuminata', 'balbisiana', 'paradisiaca'], 'issues': None}
```

The syntax everybody now uses for exchanging data is Python's dict and list syntax. Simple and correct.

- 
- 1 PyCon 2017 talk; video available at <https://youtu.be/66P5FMkWoVU?t=56>
  - 2 The original script appears in slide 41 of Martelli's "Re-learning Python" presentation. His script is actually a demonstration of `dict.setdefault`, as shown in our [Example 3-4](#).
  - 3 This is an example of using a method as a first-class function, the subject of [\[Link to Come\]](#).
  - 4 One such library is [Pingo.io](#), no longer under active development.
  - 5 The exact problem with subclassing `dict` and other built-ins is covered in [\[Link to Come\]](#).
  - 6 Dictionary views were backported to Python 2.7 and are returned by the `dict` methods `.viewkeys()`, `.viewvalues()`, and `.viewitems()`. I hope this information is of no use to you, dear reader, as Python 2.7 is now history.
  - 7 This may be interesting, but is not super important. The speed up will happen only when a set literal is evaluated, and that happens at most once per Python process—when a module is initially compiled. If you're curious, import the `dis` function from the `dis` module and use it to disassemble the bytecodes for a `set` literal—e.g. `dis('{1}')`—and a `set` call—`dis('set([1])')`
  - 8 The word "bucket" makes more sense to describe hash tables that hold more than one element per row. Python stores only one element per row, but we will stick with the colorful traditional term.
  - 9 Since I just mentioned `int`, here is a CPython implementation detail: the hash code of an `int` that fits in a machine word is the value of the `int` itself, except the hash code of `-1`, which is `-2`.
  - 10 The `hash()` built-in never returns `-1` for any Python object. If `x.__hash__()` returns `-1`, `hash(x)` returns `-2`.
  - 11 On 64-bit CPython, string hash collisions are so uncommon that I was unable to produce an example for this explanation. If you find one, let me know.
  - 12 That's how tuples are stored.
  - 13 That was before I started writing the 1st edition of *Fluent Python*, but I missed it.

- 14 It's ironic that the buckets in the hash table here do not contain hash codes, but only indexes to the `entries` array where the hash codes are. But, conceptually, the `index` array is really the hash table in this implementation, even if there are no hashes in its buckets.
- 15 Unless the class has a `__slots__` attribute, as we'll see in chapter XXX
- 16 Quoted in Butler Lampson's Turing lecture slides Principles for Computer System Design



# Chapter 4. Text versus Bytes

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at [fluentpython2e@ramalho.org](mailto:fluentpython2e@ramalho.org).

*Humans use text. Computers speak bytes.*<sup>1</sup>

—Esther Nam and Travis Fischer, Character  
Encoding and Unicode in Python

Python 3 introduced a sharp distinction between strings of human text and sequences of raw bytes. Implicit conversion of byte sequences to Unicode text is a thing of the past. This chapter deals with Unicode strings, binary sequences, and the encodings used to convert between them.

Depending on your Python programming context, a deeper understanding of Unicode may or may not be of vital importance to you. In the end, most of the issues covered in this chapter do not affect programmers who deal only with ASCII text. But even if that is your case, there is no escaping the `str` versus `byte` divide. As a bonus, you’ll find that the specialized binary sequence types provide features that the “all-purpose” Python 2 `str` type does not have.

In this chapter, we will visit the following topics:

- Characters, code points, and byte representations
- Unique features of binary sequences: `bytes`, `bytearray`, and `memoryview`
- Codecs for full Unicode and legacy character sets
- Avoiding and dealing with encoding errors
- Best practices when handling text files
- The default encoding trap and standard I/O issues
- Safe Unicode text comparisons with normalization
- Utility functions for normalization, case folding, and brute-force diacritic removal
- Proper sorting of Unicode text with `locale` and the PyUCA library
- Character metadata in the Unicode database
- Dual-mode APIs that handle `str` and `bytes`
- Building emojis from character combinations

## What's new in this chapter

Support for Unicode in Python 3 has been comprehensive and stable for a while, so the biggest change in this chapter is a new section on emojis—not because of changes in Python, but because of the growing popularity of emojis and emoji combinations. Unicode 13, released in 2020, supports more than 3000 emojis, and many of them that are built by combining Unicode characters. “Multi-character emojis” explains.

Also new in this 2<sup>nd</sup> edition is “Finding characters by name” including source code for utility for searching the Unicode database,

a great way to find circled digits and smiling cats from the command-line.

A minor change worth mentioning is the Unicode support on Windows, which is better and simpler since Python 3.6, as we'll see in ["Encoding Defaults: A Madhouse"](#).

Let's start with the not-so-new, but fundamental concepts of characters, code points, and bytes.

## Character Issues

The concept of "string" is simple enough: a string is a sequence of characters. The problem lies in the definition of "character."

In 2020, the best definition of "character" we have is a Unicode character. Accordingly, the items you get out of a Python 3 `str` are Unicode characters, just like the items of a `unicode` object in Python 2—and not the raw bytes you get from a Python 2 `str`.

The Unicode standard explicitly separates the identity of characters from specific byte representations:

- The identity of a character—its *code point*—is a number from 0 to 1,114,111 (base 10), shown in the Unicode standard as 4 to 6 hex digits with a "U+" prefix, from U+0000 to U+10FFFF. For example, the code point for the letter A is U+0041, the Euro sign is U+20AC, and the musical symbol G clef is assigned to code point U+1D11E. About 12% of the valid code points have characters assigned to them in Unicode 12.1, the standard used in Python 3.8.
- The actual bytes that represent a character depend on the *encoding* in use. An encoding is an algorithm that converts code points to byte sequences and vice versa. The code point

for the letter A (U+0041) is encoded as the single byte `\x41` in the UTF-8 encoding, or as the bytes `\x41\x00` in UTF-16LE encoding. As another example, UTF-8 requires three bytes—`\xe2\x82\xac`—to encode the Euro sign (U+20AC) but in UTF-16LE the same code point is encoded as two bytes: `\xac\x20`.

Converting from code points to bytes is *encoding*; converting from bytes to code points is *decoding*. See [Example 4-1](#).

#### *Example 4-1. Encoding and decoding*

---

```
>>> s = 'café'
>>> len(s) ❶
4
>>> b = s.encode('utf8') ❷
>>> b
b'caf\xc3\xa9' ❸
>>> len(b) ❹
5
>>> b.decode('utf8') ❺
'café'
```

- ❶ The str 'café' has four Unicode characters.
- ❷ Encode str to bytes using UTF-8 encoding.
- ❸ bytes literals have a b prefix.
- ❹ bytes b has five bytes (the code point for “é” is encoded as two bytes in UTF-8).
- ❺ Decode bytes to str using UTF-8 encoding.

## TIP

If you need a memory aid to help distinguish `.decode()` from `.encode()`, convince yourself that byte sequences can be cryptic machine core dumps while Unicode `str` objects are “human” text. Therefore, it makes sense that we *decode* bytes to `str` to get human-readable text, and we *encode* `str` to bytes for storage or transmission.

Although the Python 3 `str` is pretty much the Python 2 `unicode` type with a new name, the Python 3 `bytes` is not simply the old `str` renamed, and there is also the closely related `bytearray` type. So it is worthwhile to take a look at the binary sequence types before advancing to encoding/decoding issues.

## Byte Essentials

The new binary sequence types are unlike the Python 2 `str` in many regards. The first thing to know is that there are two basic built-in types for binary sequences: the immutable `bytes` type introduced in Python 3 and the mutable `bytearray`, added in Python 2.6.<sup>2</sup>

Each item in `bytes` or `bytearray` is an integer from 0 to 255, and not a one-character string like in the Python 2 `str`. However, a slice of a binary sequence always produces a binary sequence of the same type—including slices of length 1. See [Example 4-2](#).

*Example 4-2. A five-byte sequence as bytes and as bytearray*

---

```
>>> cafe = bytes('café', encoding='utf_8') ❶
>>> cafe
b'caf\xc3\xa9'
>>> cafe[0] ❷
99
>>> cafe[:1] ❸
```

```
b'c'
>>> cafe_arr = bytearray(cafe)
>>> cafe_arr ❹
bytearray(b'caf\xc3\xa9')
>>> cafe_arr[-1:] ❺
bytearray(b'\xa9')
```

- ❶ bytes can be built from a `str`, given an encoding.
- ❷ Each item is an integer in `range(256)`.
- ❸ Slices of bytes are also bytes—even slices of a single byte.
- ❹ There is no literal syntax for `bytearray`: they are shown as `bytearray()` with a bytes literal as argument.
- ❺ A slice of `bytearray` is also a `bytearray`.

### WARNING

The fact that `my_bytes[0]` retrieves an `int` but `my_bytes[:1]` returns a `bytes` object of length 1 may be surprising, and makes it harder to support both Python 2.7 and 3 in programs that deal with binary data. But it is consistent with many other languages and also with other Python sequence types—except for `str`, which is the only sequence type where `s[0] == s[:1]`.

Although binary sequences are really sequences of integers, their literal notation reflects the fact that ASCII text is often embedded in them. Therefore, three different displays are used, depending on each byte value:

- For bytes in the printable ASCII range—from space to `~`—the ASCII character itself is used.
- For bytes corresponding to tab, newline, carriage return, and `\`, the escape sequences `\t`, `\n`, `\r`, and `\\` are used.

- For every other byte value, a hexadecimal escape sequence is used (e.g., `\x00` is the null byte).

That is why in [Example 4-2](#) you see `b'caf\xc3\xa9'`: the first three bytes `b'caf'` are in the printable ASCII range, the last two are not.

Both `bytes` and `bytearray` support every `str` method except those that do formatting (`format`, `format_map`) and a few others that depend on Unicode data, including `casefold`, `isdecimal`, `isidentifier`, `isnumeric`, `isprintable`, and `encode`. This means that you can use familiar string methods like `endswith`, `replace`, `strip`, `translate`, `upper`, and dozens of others with binary sequences—only using `bytes` and not `str` arguments. In addition, the regular expression functions in the `re` module also work on binary sequences, if the regex is compiled from a binary sequence instead of a `str`. Since Python 3.5, the `%` operator works with binary sequences again<sup>3</sup>.

Binary sequences have a class method that `str` doesn't have, called `fromhex`, which builds a binary sequence by parsing pairs of hex digits optionally separated by spaces:

```
>>> bytes.fromhex('31 4B CE A9')
b'1K\xce\xa9'
```

The other ways of building `bytes` or `bytearray` instances are calling their constructors with:

- A `str` and an `encoding` keyword argument.
- An iterable providing items with values from 0 to 255.
- An object that implements the buffer protocol (e.g., `bytes`, `bytearray`, `memoryview`, `array.array`); this copies the bytes from the source object to the newly created binary sequence.

## WARNING

Until Python 3.5, it was also possible to call `bytes` or `bytearray` with a single integer to create a binary sequence of that size initialized with null bytes. This signature was deprecated in Python 3.5 and removed in Python 3.6. See [PEP 467 — Minor API improvements for binary sequences.](#))

Building a binary sequence from a buffer-like object is a low-level operation that may involve type casting. See a demonstration in [Example 4-3](#).

*Example 4-3. Initializing bytes from the raw data of an array*

```
>>> import array
>>> numbers = array.array('h', [-2, -1, 0, 1, 2]) ❶
>>> octets = bytes(numbers) ❷
>>> octets
b'\xfe\xff\xff\xff\x00\x00\x01\x00\x02\x00' ❸
```

- ❶ Typecode 'h' creates an array of short integers (16 bits).
- ❷ `octets` holds a copy of the bytes that make up `numbers`.
- ❸ These are the 10 bytes that represent the five short integers.

Creating a `bytes` or `bytearray` object from any buffer-like source will always copy the bytes. In contrast, `memoryview` objects let you share memory between binary data structures. To read structured information in binary sequences, the `struct` module is invaluable. We'll see it working along with `bytes` and `memoryview` in [“Structs and Memory Views”](#).

After this basic exploration of binary sequence types in Python, let's see how they are converted to/from strings.



## Basic Encoders/Decoders

The Python distribution bundles more than 100 *codecs* (encoder/decoder) for text to byte conversion and vice versa. Each codec has a name, like 'utf\_8', and often aliases, such as 'utf8', 'utf-8', and 'U8', which you can use as the `encoding` argument in functions like `open()`, `str.encode()`, `bytes.decode()`, and so on. Example 4-4 shows the same text encoded as three different byte sequences.

*Example 4-4. The string “El Niño” encoded with three codecs producing very different byte sequences*

```
>>> for codec in ['latin_1', 'utf_8', 'utf_16']:
...     print(codec, 'El Niño'.encode(codec), sep='\t')
...
latin_1 b'El Ni\xf1o'
utf_8   b'El Ni\xc3\xb1o'
utf_16  b'\xff\xfe\x00l\x00 \x00N\x00i\x00\xf1\x00o\x00'
```

Figure 4-1 demonstrates a variety of codecs generating bytes from characters like the letter “A” through the G-clef musical symbol. Note that the last three encodings are variable-length, multibyte encodings.

char.	code point	ascii	latin1	cp1252	cp437	gb2312	utf-8	utf-16le
A	U+0041	41	41	41	41	41	41	41 00
¿	U+00BF	*	BF	BF	A8	*	C2 BF	BF 00
Ã	U+00C3	*	C3	C3	*	*	C3 83	C3 00
á	U+00E1	*	E1	E1	A0	A8 A2	C3 A1	E1 00
Ω	U+03A9	*	*	*	EA	A6 B8	CE A9	A9 03
€	U+06BF	*	*	*	*	*	DA BF	BF 06
“	U+201C	*	*	93	*	A1 B0	E2 80 9C	1C 20
€	U+20AC	*	*	80	*	*	E2 82 AC	AC 20
Г	U+250C	*	*	*	DA	A9 B0	E2 94 8C	0C 25
气	U+6C14	*	*	*	*	C6 F8	E6 B0 94	14 6C
氣	U+6C23	*	*	*	*	*	E6 B0 A3	23 6C
♫	U+1D11E	*	*	*	*	*	F0 9D 84 9E	34 D8 1E DD

Figure 4-1. Twelve characters, their code points, and their byte representation (in hex) in seven different encodings (asterisks indicate that the character cannot be represented in that encoding)

All those asterisks in [Figure 4-1](#) make clear that some encodings, like ASCII and even the multibyte GB2312, cannot represent every Unicode character. The UTF encodings, however, are designed to handle every Unicode code point.

The encodings shown in [Figure 4-1](#) were chosen as a representative sample:

### *latin1 a.k.a. iso8859\_1*

Important because it is the basis for other encodings, such as cp1252 and Unicode itself (note how the `latin1` byte values appear in the cp1252 bytes and even in the code points).

### *cp1252*

A `latin1` superset by Microsoft, adding useful symbols like curly quotes and the € (euro); some Windows apps call it “ANSI,” but it was never a real ANSI standard.

### *cp437*

The original character set of the IBM PC, with box drawing characters. Incompatible with `latin1`, which appeared later.

### *gb2312*

Legacy standard to encode the simplified Chinese ideographs used in mainland China; one of several widely deployed multibyte encodings for Asian languages.

### *utf-8*

The most common 8-bit encoding on the Web, by far; as of January, 2020, [W3Techs: Usage of Character Encodings for Websites] claims that 94.7% of sites use UTF-8, up from 81.4% when I wrote this paragraph in the 1<sup>st</sup> edition of *Fluent Python* in September, 2014.

### *utf-16le*

One form of the UTF 16-bit encoding scheme; all UTF-16 encodings support code points beyond U+FFFF through escape sequences called “surrogate pairs.”

#### **WARNING**

UTF-16 superseded the original 16-bit Unicode 1.0 encoding—UCS-2—way back in 1996. UCS-2 is still used in many systems despite being deprecated since the last century because it only supports code points up to U+FFFF. As of Unicode 12.1, more than 57% of the allocated code points are above U+FFFF, including the all-important emojis.

With this overview of common encodings now complete, we move to handling issues in encoding and decoding operations.

## **Understanding Encode/Decode Problems**

Although there is a generic `UnicodeError` exception, the error reported by Python is usually more specific: either a `UnicodeEncodeError` (when converting `str` to binary sequences) or a `UnicodeDecodeError` (when reading binary sequences into `str`). Loading Python modules may also raise `SyntaxError` when the source encoding is unexpected. We'll show how to handle all of these errors in the next sections.

### TIP

The first thing to note when you get a Unicode error is the exact type of the exception. Is it a `UnicodeEncodeError`, a `UnicodeDecodeError`, or some other error (e.g., `SyntaxError`) that mentions an encoding problem? To solve the problem, you have to understand it first.

## Coping with `UnicodeEncodeError`

Most non-UTF codecs handle only a small subset of the Unicode characters. When converting text to bytes, if a character is not defined in the target encoding, `UnicodeEncodeError` will be raised, unless special handling is provided by passing an `errors` argument to the encoding method or function. The behavior of the error handlers is shown in [Example 4-5](#).

### *Example 4-5. Encoding to bytes: success and error handling*

---

```
>>> city = 'São Paulo'
>>> city.encode('utf_8') ❶
b'S\xc3\xa3o Paulo'
>>> city.encode('utf_16')
b'\xff\xfeS\x00\xe3\x00o\x00 \x00P\x00a\x00u\x00l\x00o\x00'
>>> city.encode('iso8859_1') ❷
b'S\xe3o Paulo'
>>> city.encode('cp437') ❸
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
File ".../lib/python3.4/encodings/cp437.py", line 12, in
encode
    return codecs.charmap_encode(input,errors,encoding_map)
UnicodeEncodeError: 'charmap' codec can't encode character
'\xe3' in
position 1: character maps to <undefined>
>>> city.encode('cp437', errors='ignore') ❷
b'So Paulo'
>>> city.encode('cp437', errors='replace') ❸
b'S?o Paulo'
>>> city.encode('cp437', errors='xmlcharrefreplace') ❹
b'S&#227;o Paulo'
```

- ❶ The 'utf\_?' encodings handle any str.
- ❷ 'iso8859\_1' also works for the 'São Paulo' str.
- ❸ 'cp437' can't encode the 'ã' ("a" with tilde). The default error handler—'strict'—raises UnicodeEncodeError.
- ❹ The error='ignore' handler silently skips characters that cannot be encoded; this is usually a very bad idea.
- ❺ When encoding, error='replace' substitutes unencodable characters with '?'; data is lost, but users will get a clue that something is amiss.
- ❻ 'xmlcharrefreplace' replaces unencodable characters with an XML entity.

### NOTE

The codecs error handling is extensible. You may register extra strings for the errors argument by passing a name and an error handling function to the `codecs.register_error` function. See [the `codecs.register\_error` documentation](#).

ASCII is a common subset to all the encodings that I know about, therefore encoding should always work if the text is made exclusively

of ASCII characters. Python 3.7 added a new boolean method `str.isascii()` to check whether your Unicode text is 100% pure ASCII. If it is, you should be able to encode it to bytes in any encoding without raising `UnicodeEncodeError`.

## Coping with UnicodeDecodeError

Not every byte holds a valid ASCII character, and not every byte sequence is valid UTF-8 or UTF-16; therefore, when you assume one of these encodings while converting a binary sequence to text, you will get a `UnicodeDecodeError` if unexpected bytes are found.

On the other hand, many legacy 8-bit encodings like `'cp1252'`, `'iso8859_1'`, and `'koi8_r'` are able to decode any stream of bytes, including random noise, without reporting errors. Therefore, if your program assumes the wrong 8-bit encoding, it will silently decode garbage.

### TIP

Garbled characters are known as gremlins or mojibake (文字化け—Japanese for “transformed text”).

Example 4-6 illustrates how using the wrong codec may produce gremlins or a `UnicodeDecodeError`.

### Example 4-6. Decoding from str to bytes: success and error handling

```
>>> octets = b'Montr\xe9al' ❶
>>> octets.decode('cp1252') ❷
'Montréal'
>>> octets.decode('iso8859_7') ❸
'Montrial'
>>> octets.decode('koi8_r') ❹
'MontrMał'
```

```
>>> octets.decode('utf_8') ❶
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xe9 in
position 5:
invalid continuation byte
>>> octets.decode('utf_8', errors='replace') ❷
'Montr  al'
```

- ❶ These bytes are the characters for “Montréal” encoded as `latin1`; `'\xe9'` is the byte for “  ”.
- ❷ Decoding with `'cp1252'` (Windows 1252) works because it is a proper superset of `latin1`.
- ❸ ISO-8859-7 is intended for Greek, so the `'\xe9'` byte is misinterpreted, and no error is issued.
- ❹ KOI8-R is for Russian. Now `'\xe9'` stands for the Cyrillic letter “  ”.
- ❺ The `'utf_8'` codec detects that `octets` is not valid UTF-8, and raises `UnicodeDecodeError`.
- ❻ Using `'replace'` error handling, the `\xe9` is replaced by “  ” (code point U+FFFD), the official Unicode REPLACEMENT CHARACTER intended to represent unknown characters.

## SyntaxError When Loading Modules with Unexpected Encoding

UTF-8 is the default source encoding for Python 3, just as ASCII was the default for Python 2 (starting with 2.5). If you load a `.py` module containing non-UTF-8 data and no encoding declaration, you get a message like this:

```
SyntaxError: Non-UTF-8 code starting with '\xe1' in file
ola.py on line
  1, but no encoding declared; see
http://python.org/dev/peps/pep-0263/
for details
```

Because UTF-8 is widely deployed in GNU/Linux and OSX systems, a likely scenario is opening a `.py` file created on Windows with `cp1252`. Note that this error happens even in Python for Windows, because the default encoding for Python 3 source is UTF-8 across all platforms.

To fix this problem, add a magic `coding` comment at the top of the file, as shown in [Example 4-7](#).

*Example 4-7. `ola.py`: “Hello, World!” in Portuguese*

---

```
# coding: cp1252
```

```
print('Olá, Mundo!')
```

### TIP

Now that Python 3 source code is no longer limited to ASCII and defaults to the excellent UTF-8 encoding, the best “fix” for source code in legacy encodings like `'cp1252'` is to convert them to UTF-8 already, and not bother with the `coding` comments. If your editor does not support UTF-8, it’s time to switch.



## NON-ASCII NAMES IN SOURCE CODE: SHOULD YOU USE THEM?

Python 3 allows non-ASCII identifiers in source code:

```
>>> ação = 'PBR' # ação = stock  
>>> ε = 10**-6   # ε = epsilon
```

Some people dislike the idea. The most common argument to stick with ASCII identifiers is to make it easy for everyone to read and edit code. That argument misses the point: you want your source code to be readable and editable by its intended audience, and that may not be “everyone.” If the code belongs to a multinational corporation or is open source and you want contributors from around the world, the identifiers should be in English, and then all you need is ASCII.

But if you are a teacher in Brazil, your students will find it easier to read code that uses Portuguese variable and function names, correctly spelled. And they will have no difficulty typing the cedillas and accented vowels on their localized keyboards.

Now that Python can parse Unicode names and UTF-8 is the default source encoding, I see no point in coding identifiers in Portuguese without accents, as we used to do in Python 2 out of necessity—unless you need the code to run on Python 2 also. If the names are in Portuguese, leaving out the accents won't make the code more readable to anyone.

This is my point of view as a Portuguese-speaking Brazilian, but I believe it applies across borders and cultures: choose the human language that makes the code easier to read by the team, then use the characters needed for correct spelling.

Suppose you have a text file, be it source code or poetry, but you don't know its encoding. How do you detect the actual encoding? The next section answers that with a library recommendation.

## How to Discover the Encoding of a Byte Sequence

How do you find the encoding of a byte sequence? Short answer: you can't. You must be told.

Some communication protocols and file formats, like HTTP and XML, contain headers that explicitly tell us how the content is encoded. You can be sure that some byte streams are not ASCII because they contain byte values over 127, and the way UTF-8 and UTF-16 are built also limits the possible byte sequences. But even then, you can never be 100% positive that a binary file is ASCII or UTF-8 just because certain bit patterns are not there.

However, considering that human languages also have their rules and restrictions, once you assume that a stream of bytes is human *plain text* it may be possible to sniff out its encoding using heuristics and statistics. For example, if `b'\x00'` bytes are common, it is probably a 16- or 32-bit encoding, and not an 8-bit scheme, because null characters in plain text are bugs; when the byte sequence `b'\x20\x00'` appears often, it is likely to be the space character (U+0020) in a UTF-16LE encoding, rather than the obscure U+2000 EN QUAD character—whatever that is.

That is how the package Chardet — The Universal Character Encoding Detector works to guess one of more than 30 supported encodings. Chardet is a Python library that you can use in your programs, but also includes a command-line utility, `chardetect`. Here is what it reports on the source file for this chapter:

```
$ chardetect 04-text-byte.asciidoc
04-text-byte.asciidoc: utf-8 with confidence 0.99
```

Although binary sequences of encoded text usually don't carry explicit hints of their encoding, the UTF formats may prepend a byte order mark to the textual content. That is explained next.

## BOM: A Useful Gremlin

In Example 4-4, you may have noticed a couple of extra bytes at the beginning of a UTF-16 encoded sequence. Here they are again:

```
>>> u16 = 'El Niño'.encode('utf_16')
>>> u16
b'\xff\xfeE\x00l\x00 \x00N\x00i\x00\xfa\x00o\x00'
```

The bytes are `b'\xff\xfe'`. That is a *BOM*—byte-order mark—denoting the “little-endian” byte ordering of the Intel CPU where the encoding was performed.

On a little-endian machine, for each code point the least significant byte comes first: the letter 'E', code point U+0045 (decimal 69), is encoded in byte offsets 2 and 3 as 69 and 0:

```
>>> list(u16)
[255, 254, 69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111,
0]
```

On a big-endian CPU, the encoding would be reversed; 'E' would be encoded as 0 and 69.

To avoid confusion, the UTF-16 encoding prepends the text to be encoded with the special invisible character ZERO WIDTH NO-BREAK SPACE (U+FEFF). On a little-endian system, that is encoded as `b'\xff\xfe'` (decimal 255, 254). Because, by design, there is no U+FFFE character in Unicode, the byte sequence `b'\xff\xfe'` must mean the ZERO WIDTH NO-BREAK SPACE on a little-endian encoding, so the codec knows which byte ordering to use.

There is a variant of UTF-16—UTF-16LE—that is explicitly little-endian, and another one explicitly big-endian, UTF-16BE. If you use them, a BOM is not generated:

```
>>> u16le = 'El Niño'.encode('utf_16le')
>>> list(u16le)
[69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111, 0]
>>> u16be = 'El Niño'.encode('utf_16be')
```

```
>>> list(u16be)
[0, 69, 0, 108, 0, 32, 0, 78, 0, 105, 0, 241, 0, 111]
```

If present, the BOM is supposed to be filtered by the UTF-16 codec, so that you only get the actual text contents of the file without the leading ZERO WIDTH NO-BREAK SPACE. The Unicode standard says that if a file is UTF-16 and has no BOM, it should be assumed to be UTF-16BE (big-endian). However, the Intel x86 architecture is little-endian, so there is plenty of little-endian UTF-16 with no BOM in the wild.

This whole issue of endianness only affects encodings that use words of more than one byte, like UTF-16 and UTF-32. One big advantage of UTF-8 is that it produces the same byte sequence regardless of machine endianness, so no BOM is needed. Nevertheless, some Windows applications (notably Notepad) add the BOM to UTF-8 files anyway—and Excel depends on the BOM to detect a UTF-8 file, otherwise it assumes the content is encoded with a Windows code page. The character U+FEFF encoded in UTF-8 is the three-byte sequence `b'\xef\xbb\xbf'`. So if a file starts with those three bytes, it is likely to be a UTF-8 file with a BOM. However, Python does not automatically assume a file is UTF-8 just because it starts with `b'\xef\xbb\xbf'`.

We now move on to handling text files in Python 3.

## Handling Text Files

The best practice for handling text I/O is the “Unicode sandwich” (Figure 4-2).<sup>4</sup> This means that `bytes` should be decoded to `str` as early as possible on input (e.g., when opening a file for reading). The “filling” of the sandwich is the business logic of your program, where text handling is done exclusively on `str` objects. You should never be encoding or decoding in the middle of other processing. On output,

the `str` are encoded to `bytes` as late as possible. Most web frameworks work like that, and we rarely touch `bytes` when using them. In Django, for example, your views should output Unicode `str`; Django itself takes care of encoding the response to `bytes`, using UTF-8 by default.

## The Unicode sandwich



Figure 4-2. Unicode sandwich: current best practice for text processing

Python 3 makes it easier to follow the advice of the Unicode sandwich, because the `open` built-in does the necessary decoding when reading and encoding when writing files in text mode, so all you get from `my_file.read()` and pass to `my_file.write(text)` are `str` objects.<sup>5</sup>

Therefore, using text files is apparently simple. But if you rely on default encodings you will get bitten.

Consider the console session in [Example 4-8](#). Can you spot the bug?

*Example 4-8. A platform encoding issue (if you try this on your machine, you may or may not see the problem)*

```
>>> open('cafe.txt', 'w', encoding='utf_8').write('café')
4
>>> open('cafe.txt').read()
'cafÃ©'
```

The bug: I specified UTF-8 encoding when writing the file but failed to do so when reading it, so Python assumed Windows default file encoding—code page 1252—and the trailing bytes in the file were decoded as characters 'Ã©' instead of 'é'.

I ran [Example 4-8](#) on Python 3.8.1, 64 bits, on Windows 10 (build 18363). The same statements running on recent GNU/Linux or Mac OSX work perfectly well because their default encoding is UTF-8, giving the false impression that everything is fine. If the encoding argument was omitted when opening the file to write, the locale default encoding would be used, and we'd read the file correctly using the same encoding. But then this script would generate files with different byte contents depending on the platform or even depending on locale settings in the same platform, creating compatibility problems.

### TIP

Code that has to run on multiple machines or on multiple occasions should never depend on encoding defaults. Always pass an explicit `encoding=` argument when opening text files, because the default may change from one machine to the next, or from one day to the next.

A curious detail in [Example 4-8](#) is that the `write` function in the first statement reports that four characters were written, but in the next line five characters are read. [Example 4-9](#) is an extended version of [Example 4-8](#), explaining that and other details.

*Example 4-9. Closer inspection of [Example 4-8](#) running on Windows reveals the bug and how to fix it*

---

```
>>> fp = open('cafe.txt', 'w', encoding='utf_8')
>>> fp ❶
<_io.TextIOWrapper name='cafe.txt' mode='w' encoding='utf_8'>
```

```

>>> fp.write('café') ❷
4
>>> fp.close()
>>> import os
>>> os.stat('cafe.txt').st_size ❸
5
>>> fp2 = open('cafe.txt')
>>> fp2 ❹
<_io.TextIOWrapper name='cafe.txt' mode='r' encoding='cp1252'>
>>> fp2.encoding ❺
'cp1252'
>>> fp2.read() ❻
'cafÃ©'
>>> fp3 = open('cafe.txt', encoding='utf_8') ❼
>>> fp3
<_io.TextIOWrapper name='cafe.txt' mode='r' encoding='utf_8'>
>>> fp3.read() ❽
'café'
>>> fp4 = open('cafe.txt', 'rb') ❾
>>> fp4 ❿
<_io.BufferedReader name='cafe.txt'>
>>> fp4.read() ⓫
b'caf\xc3\xa9'

```

- ❶ By default, open uses text mode and returns a TextIOWrapper object with a specific encoding.
- ❷ The write method on a TextIOWrapper returns the number of Unicode characters written.
- ❸ os.stat says the file has 5 bytes; UTF-8 encodes 'é' as 2 bytes, 0xc3 and 0xa9.
- ❹ Opening a text file with no explicit encoding returns a TextIOWrapper with the encoding set to a default from the locale.
- ❺ A TextIOWrapper object has an encoding attribute that you can inspect: cp1252 in this case.
- ❻ In the Windows cp1252 encoding, the byte 0xc3 is an “Ã” (A with tilde) and 0xa9 is the copyright sign.
- ❼ Opening the same file with the correct encoding.

- ⑧ The expected result: the same four Unicode characters for 'café'.
- ⑨ The 'rb' flag opens a file for reading in binary mode.
- ⑩ The returned object is a `BufferedReader` and not a `TextIOWrapper`.
- ⑪ Reading that returns bytes, as expected.

### TIP

Do not open text files in binary mode unless you need to analyze the file contents to determine the encoding—even then, you should be using Chardet instead of reinventing the wheel (see “[How to Discover the Encoding of a Byte Sequence](#)”). Ordinary code should only use binary mode to open binary files, like raster images.

The problem in [Example 4-9](#) has to do with relying on a default setting while opening a text file. There are several sources for such defaults, as the next section shows.

## Encoding Defaults: A Madhouse

Several settings affect the encoding defaults for I/O in Python. See the `default_encodings.py` script in [Example 4-10](#).

*Example 4-10. Exploring encoding defaults*

---

```
import sys, locale

expressions = """
    locale.getpreferredencoding()
    type(my_file)
    my_file.encoding
    sys.stdout.isatty()
    sys.stdout.encoding
    sys.stdin.isatty()
```



```

        sys.stdin.encoding
        sys.stderr.isatty()
        sys.stderr.encoding
        sys.getdefaultencoding()
        sys.getfilesystemencoding()
    """

my_file = open('dummy', 'w')

for expression in expressions.split():
    value = eval(expression)
    print(expression.rjust(30), '->', repr(value))

```

The output of [Example 4-10](#) on GNU/Linux (Ubuntu 14.04 to 19.10) and MacOS (10.9 to 10.14) is identical, showing that UTF-8 is used everywhere in these systems:

```

$ python3 default_encodings.py
locale.getpreferredencoding() -> 'UTF-8'
    type(my_file) -> <class '_io.TextIOWrapper'>
    my_file.encoding -> 'UTF-8'
sys.stdout.isatty() -> True
sys.stdout.encoding -> 'utf-8'
sys.stdin.isatty() -> True
sys.stdin.encoding -> 'utf-8'
sys.stderr.isatty() -> True
sys.stderr.encoding -> 'utf-8'
sys.getdefaultencoding() -> 'utf-8'
sys.getfilesystemencoding() -> 'utf-8'

```

On Windows, however, the output is [Example 4-11](#).

*Example 4-11. Default encodings on Windows 10 PowerShell (output is the same on cmd.exe)*

---

```

> chcp ❶
Active code page: 437
> python default_encodings.py ❷
    locale.getpreferredencoding() -> 'cp1252' ❸

```

```
type(my_file) -> <class '_io.TextIOWrapper'>
my_file.encoding -> 'cp1252' ❹
sys.stdout.isatty() -> True ❺
sys.stdout.encoding -> 'utf-8' ❻
sys.stdin.isatty() -> True
sys.stdin.encoding -> 'utf-8'
sys.stderr.isatty() -> True
sys.stderr.encoding -> 'utf-8'
sys.getdefaultencoding() -> 'utf-8'
sys.getfilesystemencoding() -> 'utf-8'
```

- ❶ `chcp` shows the active code page for the console: 437.
- ❷ Running `default_encodings.py` with output to console.
- ❸ `locale.getpreferredencoding()` is the most important setting.
- ❹ Text files use `locale.getpreferredencoding()` by default.
- ❺ The output is going to the console, so `sys.stdout.isatty()` is `True`.
- ❻ Now, `sys.stdout.encoding` is not the same as the console code page reported by `chcp`!

Unicode support in Windows itself, and in Python for Windows, got better since I wrote about this in the 1<sup>st</sup> edition of *\_Fluent Python*. [Example 4-11](#) used to report four different encodings in Python 3.4 on Windows 7. The encodings for `stdout`, `stdin`, and `stderr` used to be the same as the active code page reported by the `chcp` command, but now they're all `utf-8` thanks to [PEP 528: Change Windows console encoding to UTF-8](#) implemented in Python 3.6, and Unicode support in PowerShell and `cmd.exe` (since Windows 1809 from October, 2018<sup>6</sup>). It's weird that `chcp` and `sys.stdout.encoding` say different things when `stdout` is writing to the console, but it's great that now we can print Unicode strings without encoding errors on Windows—unless the user redirects output to a file, as we'll soon see. That does not mean all your favorite emojis will appear in the console: that also depends on the font the console is using.

Another change was PEP 529: Change Windows filesystem encoding to UTF-8, also implemented in Python 3.6, which changed the file system encoding (used to represent names of directories and files) from Microsoft's proprietary MBCS to UTF-8.

However, if the output of [Example 4-10](#) is redirected to a file, like this:

```
Z:\>python default_encodings.py > encodings.log
```

Then, the value of `sys.stdout.isatty()` becomes `False`, and `sys.stdout.encoding` is set by `locale.getpreferredencoding()`, `'cp1252'` in that machine—but `sys.stdin.encoding` and `sys.stderr.encoding` remain `utf-8`.

This means that a script like [Example 4-12](#) works when printing to the console, but may break when output is redirected to a file.

*Example 4-12. `stdout_check.py`*

---

```
import sys
from unicodedata import name

print(sys.version)
print()
print('sys.stdout.isatty():', sys.stdout.isatty())
print('sys.stdout.encoding:', sys.stdout.encoding)
print()

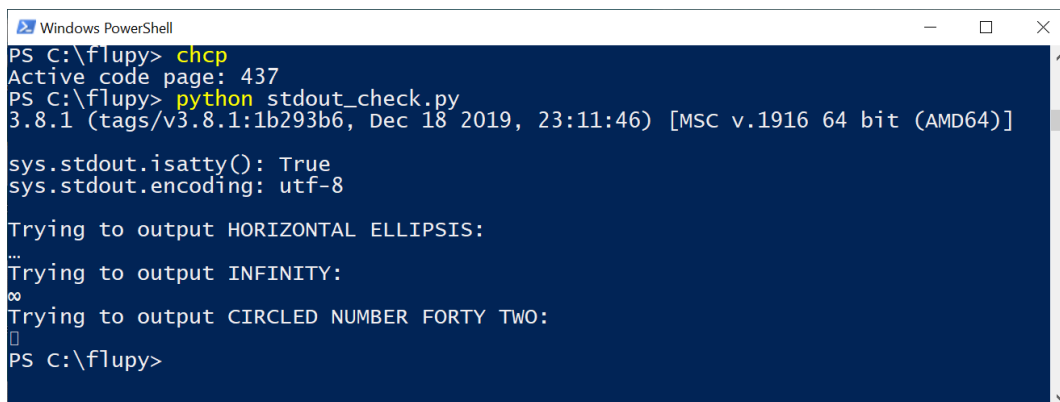
test_chars = [
    '\u2026', # HORIZONTAL ELLIPSIS (in cp1252)
    '\u221E', # INFINITY (in cp437)
    '\u32B7', # CIRCLED NUMBER FORTY TWO
]

for char in test_chars:
    print(f'Trying to output {name(char)}:')
    print(char)
```

Example 4-12 displays the result of `sys.stdout.isatty()`, the value of `sys.stdout.encoding`, and these three characters:

- '...' HORIZONTAL ELLIPSIS (U+2026)--exists in CP 1252 but not in CP 437
- '∞' INFINITY (U+221E)--exists in CP 437 but not in CP 1252
- '④' CIRCLED NUMBER FORTY TWO (U+2460)--doesn't exist in CP 1252 or CP 437

When I run `stdout_check.py` on PowerShell or `cmd.exe`, it works as captured in Figure 4-3.



```
Windows PowerShell
PS C:\flupy> chcp
Active code page: 437
PS C:\flupy> python stdout_check.py
3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 23:11:46) [MSC v.1916 64 bit (AMD64)]

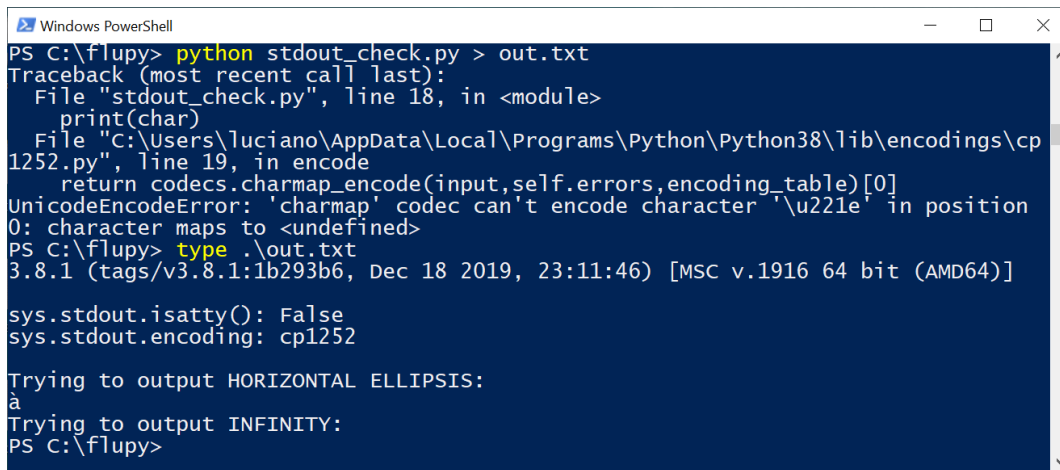
sys.stdout.isatty(): True
sys.stdout.encoding: utf-8

Trying to output HORIZONTAL ELLIPSIS:
...
Trying to output INFINITY:
∞
Trying to output CIRCLED NUMBER FORTY TWO:
□
PS C:\flupy>
```

Figure 4-3. Running `stdout_check.py` on PowerShell.

Despite `chcp` reporting the active code as 437, `sys.stdout.encoding` is UTF-8, so the HORIZONTAL ELLIPSIS and INFINITY both output correctly. The CIRCLED NUMBER FORTY TWO is replaced by a rectangle, but no error is raised. Presumably it is recognized as a valid character, but the console font doesn't have the glyph to display it.

However, when I redirect the output of `stdout_check.py` to a file, I get Figure 4-4.



```
PS C:\flupy> python stdout_check.py > out.txt
Traceback (most recent call last):
  File "stdout_check.py", line 18, in <module>
    print(char)
  File "C:\Users\luciano\AppData\Local\Programs\Python\Python38\lib\encodings\cp
1252.py", line 19, in encode
    return codecs.charmap_encode(input,self.errors,encoding_table)[0]
UnicodeEncodeError: 'charmap' codec can't encode character '\u221e' in position
0: character maps to <undefined>
PS C:\flupy> type .\out.txt
3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 23:11:46) [MSC v.1916 64 bit (AMD64)]

sys.stdout.isatty(): False
sys.stdout.encoding: cp1252

Trying to output HORIZONTAL ELLIPSIS:
à
Trying to output INFINITY:
PS C:\flupy>
```

Figure 4-4. Running *stdout\_check.py* on PowerShell, redirecting output.

The first problem demonstrated by Figure 4-4 is the `UnicodeEncodeError` mentioning character `'\u221e'`, because `sys.stdout.encoding` is `'cp1252'`--a code page that doesn't have the INFINITY character.

Then, inspecting the partially-written `out.txt`, I get two surprises:

1. Reading `out.txt` with the `type` command—or a Windows editor like VS Code or Sublime Text—shows that instead of HORIZONTAL ELLIPSIS, I got `'à'` (LATIN SMALL LETTER A WITH GRAVE). As it turns out, the byte value `0x85` in CP 1252 means `'...'`, but in CP 437 the same byte value represents `'à'`. So it seems the active code page does matter, not in a sensible or useful way, but as partial explanation of a bad Unicode experience.
2. `out.txt` was written with the UTF-16 LE encoding. This would be good, as UTF encodings support all Unicode characters—if it wasn't for the unfortunate replacement of `'...'` with `'à'`.

## NOTE

I used a laptop configured for the US market, running Windows 10 OEM to run these experiments. Windows versions localized for other countries may have different encoding configurations. For example, in Brazil the Windows console uses code page 850 by default—not 437.

To wrap up this maddening issue of default encodings, let's give a final look at the different encodings in [Example 4-11](#):

- If you omit the encoding argument when opening a file, the default is given by `locale.getpreferredencoding()` ('cp1252' in [Example 4-11](#)).
- The encoding of `sys.stdout|stdin|stderr` used to be set by the `PYTHONIOENCODING` environment variable before Python 3.6—now that variable is ignored, unless `PYTHONLEGACYWINDOWSSTDIO` is set to a non-empty string. Otherwise, the encoding for standard I/O is UTF-8 for interactive I/O, or defined by `locale.getpreferredencoding()` if the output/input is redirected to/from a file.
- `sys.getdefaultencoding()` is used internally by Python in implicit conversions of binary data to/from `str`; this happens less often in Python 3, but still happens.<sup>7</sup> Changing this setting is not supported.<sup>8</sup>
- `sys.getfilesystemencoding()` is used to encode/decode filenames (not file contents). It is used when `open()` gets a `str` argument for the filename; if the filename is given as a `bytes` argument, it is passed unchanged to the OS API. Before Python 3.6, this was MBCS on Windows, now it's

UTF-8. (On this topic, a useful answer on StackOverflow is [“Difference between MBCS and UTF-8 on Windows”](#).)

### NOTE

On GNU/Linux and OSX all of these encodings are set to UTF-8 by default, and have been for several years, so I/O handles all Unicode characters. On Windows, not only are different encodings used in the same system, but they are usually code pages like 'cp850' or 'cp1252' that support only ASCII with 127 additional characters that are not the same from one encoding to the other. Therefore, Windows users are far more likely to face encoding errors unless they are extra careful.

To summarize, the most important encoding setting is that returned by `locale.getpreferredencoding()`: it is the default for opening text files and for `sys.stdout/stdin/stderr` when they are redirected to files. However, the [documentation](#) reads (in part):

*`locale.getpreferredencoding(do_setlocale=True)`*

*Return the encoding used for text data, according to user preferences. User preferences are expressed differently on different systems, and might not be available programmatically on some systems, so this function only returns a guess. [...]*

Therefore, the best advice about encoding defaults is: do not rely on them.

You will avoid a lot of pain if you follow the advice of the Unicode sandwich and always are explicit about the encodings in your programs. Unfortunately, Unicode is painful even if you get your bytes correctly converted to `str`. The next two sections cover subjects that are simple in ASCII-land, but get quite complex on

planet Unicode: text normalization (i.e., converting text to a uniform representation for comparisons) and sorting.

## Normalizing Unicode for Saner Comparisons

String comparisons are complicated by the fact that Unicode has combining characters: diacritics and other marks that attach to the preceding character, appearing as one when printed.

For example, the word “café” may be composed in two ways, using four or five code points, but the result looks exactly the same:

```
>>> s1 = 'café'
>>> s2 = 'cafe\u0301'
>>> s1, s2
('café', 'café')
>>> len(s1), len(s2)
(4, 5)
>>> s1 == s2
False
```

The code point U+0301 is the COMBINING ACUTE ACCENT. Using it after “e” renders “é”. In the Unicode standard, sequences like 'é' and 'e\u0301' are called “canonical equivalents,” and applications are supposed to treat them as the same. But Python sees two different sequences of code points, and considers them not equal.

The solution is to use Unicode normalization, provided by the `unicodedata.normalize` function. The first argument to that function is one of four strings: 'NFC', 'NFD', 'NFKC', and 'NFKD'. Let's start with the first two.

Normalization Form C (NFC) composes the code points to produce the shortest equivalent string, while NFD decomposes, expanding composed characters into base characters and separate combining



characters. Both of these normalizations make comparisons work as expected:

```
>>> from unicodedata import normalize
>>> s1 = 'café' # composed "e" with acute accent
>>> s2 = 'cafe\u0301' # decomposed "e" and acute accent
>>> len(s1), len(s2)
(4, 5)
>>> len(normalize('NFC', s1)), len(normalize('NFC', s2))
(4, 4)
>>> len(normalize('NFD', s1)), len(normalize('NFD', s2))
(5, 5)
>>> normalize('NFC', s1) == normalize('NFC', s2)
True
>>> normalize('NFD', s1) == normalize('NFD', s2)
True
```

Western keyboards usually generate composed characters, so text typed by users will be in NFC by default. However, to be safe, it may be good to sanitize strings with `normalize('NFC', user_text)` before saving. NFC is also the normalization form recommended by the W3C in [Character Model for the World Wide Web: String Matching and Searching](#).

Some single characters are normalized by NFC into another single character. The symbol for the ohm ( $\Omega$ ) unit of electrical resistance is normalized to the Greek uppercase omega. They are visually identical, but they compare unequal so it is essential to normalize to avoid surprises:

```
>>> from unicodedata import normalize, name
>>> ohm = '\u2126'
>>> name(ohm)
'OHM SIGN'
>>> ohm_c = normalize('NFC', ohm)
>>> name(ohm_c)
'GREEK CAPITAL LETTER OMEGA'
```

```
>>> ohm == ohm_c
False
>>> normalize('NFC', ohm) == normalize('NFC', ohm_c)
True
```

In the acronyms for the other two normalization forms—NFKC and NFKD—the letter K stands for “compatibility.” These are stronger forms of normalization, affecting the so-called “compatibility characters.” Although one goal of Unicode is to have a single “canonical” code point for each character, some characters appear more than once for compatibility with preexisting standards. For example, the micro sign, 'μ' (U+00B5), was added to Unicode to support round-trip conversion to `latin1`, even though the same character is part of the Greek alphabet with code point U+03BC (GREEK SMALL LETTER MU). So, the micro sign is considered a “compatibility character.”

In the NFKC and NFKD forms, each compatibility character is replaced by a “compatibility decomposition” of one or more characters that are considered a “preferred” representation, even if there is some formatting loss—ideally, the formatting should be the responsibility of external markup, not part of Unicode. To exemplify, the compatibility decomposition of the one half fraction '½' (U+00BD) is the sequence of three characters '1/2', and the compatibility decomposition of the micro sign 'μ' (U+00B5) is the lowercase mu 'μ' (U+03BC).<sup>9</sup>

Here is how the NFKC works in practice:

```
>>> from unicodedata import normalize, name
>>> half = '½'
>>> normalize('NFKC', half)
'1/2'
>>> four_squared = '4²'
>>> normalize('NFKC', four_squared)
'42'
```

```
>>> micro = 'μ'
>>> micro_kc = normalize('NFKC', micro)
>>> micro, micro_kc
('μ', 'μ')
>>> ord(micro), ord(micro_kc)
(181, 956)
>>> name(micro), name(micro_kc)
('MICRO SIGN', 'GREEK SMALL LETTER MU')
```

Although '½' is a reasonable substitute for '½', and the micro sign is really a lowercase Greek mu, converting '4²' to '42' changes the meaning. An application could store '4²' as '4<sup>2</sup>', but the `normalize` function knows nothing about formatting. Therefore, NFKC or NFKD may lose or distort information, but they can produce convenient intermediate representations for searching and indexing: users may be pleased that a search for '½ inch' also finds documents containing '½ inch'.

### WARNING

NFKC and NFKD normalization should be applied with care and only in special cases—e.g., search and indexing—and not for permanent storage, because these transformations cause data loss.

When preparing text for searching or indexing, another operation is useful: case folding, our next subject.

## Case Folding

Case folding is essentially converting all text to lowercase, with some additional transformations. It is supported by the `str.casefold()` method since Python 3.3.

For any string `s` containing only `latin1` characters, `s.casefold()` produces the same result as `s.lower()`, with only two exceptions—the micro sign 'μ' is changed to the Greek lowercase mu (which looks the same in most fonts) and the German Eszett or “sharp s” (ß) becomes “ss”:

```
>>> micro = 'μ'
>>> name(micro)
'MICRO SIGN'
>>> micro_cf = micro.casefold()
>>> name(micro_cf)
'GREEK SMALL LETTER MU'
>>> micro, micro_cf
('μ', 'μ')
>>> eszett = 'ß'
>>> name(eszett)
'LATIN SMALL LETTER SHARP S'
>>> eszett_cf = eszett.casefold()
>>> eszett, eszett_cf
('ß', 'ss')
```

There are nearly 300 code points for which `str.casefold()` and `str.lower()` return different results.

As usual with anything related to Unicode, case folding is a complicated issue with plenty of linguistic special cases, but the Python core team made an effort to provide a solution that hopefully works for most users.

In the next couple of sections, we’ll put our normalization knowledge to use developing utility functions.

## Utility Functions for Normalized Text Matching

As we’ve seen, NFC and NFD are safe to use and allow sensible comparisons between Unicode strings. NFC is the best normalized

form for most applications. `str.casefold()` is the way to go for case-insensitive comparisons.

If you work with text in many languages, a pair of functions like `nfc_equal` and `fold_equal` in [Example 4-13](#) are useful additions to your toolbox.

---

*Example 4-13. `normeq.py`: normalized Unicode string comparison*

```
"""
```

*Utility functions for normalized Unicode string comparison.*

*Using Normal Form C, case sensitive:*

```
>>> s1 = 'café'
>>> s2 = 'cafe\u0301'
>>> s1 == s2
False
>>> nfc_equal(s1, s2)
True
>>> nfc_equal('A', 'a')
False
```

*Using Normal Form C with case folding:*

```
>>> s3 = 'Straße'
>>> s4 = 'strasse'
>>> s3 == s4
False
>>> nfc_equal(s3, s4)
False
>>> fold_equal(s3, s4)
True
>>> fold_equal(s1, s2)
True
>>> fold_equal('A', 'a')
True
```

```
"""
```

```
from unicodedata import normalize

def nfc_equal(str1, str2):
    return normalize('NFC', str1) == normalize('NFC', str2)

def fold_equal(str1, str2):
    return (normalize('NFC', str1).casefold() ==
            normalize('NFC', str2).casefold())
```

Beyond Unicode normalization and case folding—which are both part of the Unicode standard—sometimes it makes sense to apply deeper transformations, like changing 'café' into 'cafe'. We'll see when and how in the next section.

## Extreme “Normalization”: Taking Out Diacritics

The Google Search secret sauce involves many tricks, but one of them apparently is ignoring diacritics (e.g., accents, cedillas, etc.), at least in some contexts. Removing diacritics is not a proper form of normalization because it often changes the meaning of words and may produce false positives when searching. But it helps coping with some facts of life: people sometimes are lazy or ignorant about the correct use of diacritics, and spelling rules change over time, meaning that accents come and go in living languages.

Outside of searching, getting rid of diacritics also makes for more readable URLs, at least in Latin-based languages. Take a look at the URL for the Wikipedia article about the city of São Paulo:

```
http://en.wikipedia.org/wiki/S%C3%A3o_Paulo
```

The %C3%A3 part is the URL-escaped, UTF-8 rendering of the single letter “ã” (“a” with tilde). The following is much friendlier, even if it is not the right spelling:

```
http://en.wikipedia.org/wiki/Sao_Paulo
```

To remove all diacritics from a `str`, you can use a function like Example 4-14.

*Example 4-14. Function to remove all combining marks (module `sanitize.py`).*

```
import unicodedata
import string

def shave_marks(txt):
    """Remove all diacritic marks"""
    norm_txt = unicodedata.normalize('NFD', txt) ❶
    shaved = ''.join(c for c in norm_txt
                     if not unicodedata.combining(c)) ❷
    return unicodedata.normalize('NFC', shaved) ❸
```

- ❶ Decompose all characters into base characters and combining marks.
- ❷ Filter out all combining marks.
- ❸ Recompose all characters.

Example 4-15 shows a couple of uses of `shave_marks`.

*Example 4-15. Two examples using `shave_marks` from Example 4-14*

```
>>> order = '“Herr Voß: • ½ cup of Ætker™ caffè latte • bowl of  
açai.”'  
>>> shave_marks(order)  
'“Herr Voß: • ½ cup of Ætker™ caffè latte • bowl of acai.”' ❶  
>>> Greek = 'Ζέφυρος, Ζέφiro'  
>>> shave_marks(Greek)  
'Ζεφυρος, Zefiro' ❷
```

- ❶ Only the letters “è”, “ç”, and “í” were replaced.
- ❷ Both “ξ” and “é” were replaced.

The function `shave_marks` from [Example 4-14](#) works all right, but maybe it goes too far. Often the reason to remove diacritics is to change Latin text to pure ASCII, but `shave_marks` also changes non-Latin characters—like Greek letters—which will never become ASCII just by losing their accents. So it makes sense to analyze each base character and to remove attached marks only if the base character is a letter from the Latin alphabet. This is what [Example 4-16](#) does.

*Example 4-16. Function to remove combining marks from Latin characters (import statements are omitted as this is part of the `sanitize.py` module from [Example 4-14](#))*

---

```
def shave_marks_latin(txt):  
    """Remove all diacritic marks from Latin base characters"""  
    norm_txt = unicodedata.normalize('NFD', txt) ❶  
    latin_base = False  
    keepers = []  
    for c in norm_txt:  
        if unicodedata.combining(c) and latin_base: ❷  
            continue # ignore diacritic on Latin base char  
        keepers.append(c) ❸  
        # if it isn't combining char, it's a new base char  
        if not unicodedata.combining(c): ❹  
            latin_base = c in string.ascii_letters  
    shaved = ''.join(keepers)  
    return unicodedata.normalize('NFC', shaved) ❺
```

- ❶ Decompose all characters into base characters and combining marks.
- ❷ Skip over combining marks when base character is Latin.
- ❸ Otherwise, keep current character.
- ❹ Detect new base character and determine if it's Latin.
- ❺ Recompose all characters.

An even more radical step would be to replace common symbols in Western texts (e.g., curly quotes, em dashes, bullets, etc.) into ASCII equivalents. This is what the function `asciiize` does in [Example 4-17](#).



```
single_map = str.maketrans(" ", f„†^‘ ’“”•—~>""", ❶  
                            """"f"*^<' '"_-~>""")  
  
multi_map = str.maketrans({ ❷  
    '€': '<euro>',  
    '…': '...',  
    'Œ': 'OE',  
    '™': '(TM)',  
    'æ': 'oe',  
    '%': '<per mille>',  
    '‡': '**',  
})  
  
multi_map.update(single_map) ❸  
  
def dewinize(txt):  
    """Replace Win1252 symbols with ASCII chars or sequences"""  
    return txt.translate(multi_map) ❹  
  
def asciize(txt):  
    no_marks = shave_marks_latin(dewinize(txt)) ❺  
    no_marks = no_marks.replace('ß', 'ss') ❻  
    return unicodedata.normalize('NFKC', no_marks) ❼
```

- 1 Build mapping table for char-to-char replacement.
- 2 Build mapping table for char-to-string replacement.
- 3 Merge mapping tables.
- 4 `dewinize` does not affect ASCII or `latin1` text, only the Microsoft additions in to `latin1` in `cp1252`.
- 5 Apply `dewinize` and remove diacritical marks.
- 6 Replace the Eszett with “ss” (we are not using case fold here because we want to preserve the case).
- 7 Apply NFKC normalization to compose characters with their compatibility code points.

Example 4-18 shows `asciiize` in use.

*Example 4-18. Two examples using `asciiize` from Example 4-17*

```
>>> order = '“Herr Voß: • ½ cup of Etker™ caffè latte • bowl of  
açai.”’'  
>>> dewinize(order)  
'"Herr Voß: - ½ cup of Etker(TM) caffè latte - bowl of açai."'  
❶  
>>> asciiize(order)  
'"Herr Voss: - ½ cup of Etker(TM) caffè latte - bowl of acai."'  
❷
```

- ❶ `dewinize` replaces curly quotes, bullets, and <sup>TM</sup> (trademark symbol).
- ❷ `asciiize` applies `dewinize`, drops diacritics, and replaces the 'ß'.

### WARNING

Different languages have their own rules for removing diacritics. For example, Germans change the 'ü' into 'ue'. Our `asciiize` function is not as refined, so it may or not be suitable for your language. It works acceptably for Portuguese, though.

To summarize, the functions in `sanitize.py` go way beyond standard normalization and perform deep surgery on the text, with a good chance of changing its meaning. Only you can decide whether to go so far, knowing the target language, your users, and how the transformed text will be used.

This wraps up our discussion of normalizing Unicode text.

The next Unicode matter to sort out is... sorting.

## Sorting Unicode Text

Python sorts sequences of any type by comparing the items in each sequence one by one. For strings, this means comparing the code points. Unfortunately, this produces unacceptable results for anyone who uses non-ASCII characters.

Consider sorting a list of fruits grown in Brazil:

```
>>> fruits = ['caju', 'atemoia', 'cajá', 'açaí', 'acerola']
>>> sorted(fruits)
['acerola', 'atemoia', 'açaí', 'caju', 'cajá']
```

Sorting rules vary for different locales, but in Portuguese and many languages that use the Latin alphabet, accents and cedillas rarely make a difference when sorting.<sup>10</sup> So “cajá” is sorted as “caja,” and must come before “caju.”

The sorted `fruits` list should be:

```
['açaí', 'acerola', 'atemoia', 'cajá', 'caju']
```

The standard way to sort non-ASCII text in Python is to use the `locale.strxfrm` function which, according to the [locale module docs](#), “transforms a string to one that can be used in locale-aware comparisons.”

To enable `locale.strxfrm`, you must first set a suitable locale for your application, and pray that the OS supports it. The sequence of commands in [Example 4-19](#) may work for you.

*Example 4-19. `locale_sort.py`: using the `locale.strxfrm` function as sort key*

---

```
include::code/04-text-byte/locale_sort.py
```

Running [Example 4-19](#) on GNU/Linux (Ubuntu 19.10) with the `pt_BR.UTF-8` locale installed, I get this result:

```
'pt_BR.UTF-8'  
['açai', 'acerola', 'atemoia', 'cajá', 'caju']
```

So you need to call `setlocale(LC_COLLATE, «your_locale»)` before using `locale.strxfrm` as the key when sorting.

There are some caveats, though:

- Because locale settings are global, calling `setlocale` in a library is not recommended. Your application or framework should set the locale when the process starts, and should not change it afterwards.
- The locale must be installed on the OS, otherwise `setlocale` raises a `locale.Error: unsupported locale setting` exception.
- You must know how to spell the locale name. They are pretty much standardized in Unix derivatives as `'language_code.encoding'`, but on Windows the syntax is more complicated: `Language Name-Language Variant_Region Name.codepage`. Note that the Language Name, Language Variant, and Region Name parts can have spaces inside them, but the parts after the first are prefixed with special different characters: a hyphen, an underline character, and a dot. All parts seem to be optional except the language name. For example: `English_United States.850` means Language Name “English”, region “United States”, and code page “850”. The language and region names Windows understands are listed in the MSDN article [Language Identifier Constants and Strings](#), while [Code Page Identifiers](#) lists the numbers for the last part.<sup>11</sup>

- The locale must be correctly implemented by the makers of the OS. I was successful on Ubuntu 19.10, but not on MacOS 10.14. On MacOS, the call `setlocale(LC_COLLATE, 'pt_BR.UTF-8')` returns the string `'pt_BR.UTF-8'` with no complaints. But `sorted(fruits, key=locale.strxfrm)` produced the same incorrect result as `sorted(fruits)` did. I also tried the `fr_FR`, `es_ES`, and `de_DE` locales on OSX, but `locale.strxfrm` never did its job.<sup>12</sup>

So the standard library solution to internationalized sorting works, but seems to be well supported only on GNU/Linux (perhaps also on Windows, if you are an expert). Even then, it depends on locale settings, creating deployment headaches.

Fortunately, there is a simpler solution: the PyUCA library, available on *PyPI*.

## Sorting with the Unicode Collation Algorithm

James Tauber, prolific Django contributor, must have felt the pain and created *PyUCA*, a pure-Python implementation of the Unicode Collation Algorithm (UCA). [Example 4-20](#) shows how easy it is to use.

*Example 4-20. Using the `pyuca.Collator.sort_key` method*

---

```
>>> import pyuca
>>> coll = pyuca.Collator()
>>> fruits = ['caju', 'atemoia', 'cajá', 'açaí', 'acerola']
>>> sorted_fruits = sorted(fruits, key=coll.sort_key)
>>> sorted_fruits
['açaí', 'acerola', 'atemoia', 'cajá', 'caju']
```

This is friendly and just works. I tested it on GNU/Linux, OSX, and Windows. Only Python 3.X is supported at this time.

PyUCA does not take the locale into account. If you need to customize the sorting, you can provide the path to a custom collation table to the `Collator()` constructor. Out of the box, it uses `allkeys.txt`, which is bundled with the project. That's just a copy of the [Default Unicode Collation Element Table](#) from Unicode.org.

By the way, that table is one of the many that comprise the Unicode database, our next subject.

## The Unicode Database

The Unicode standard provides an entire database—in the form of several structured text files—that includes not only the table mapping code points to character names, but also metadata about the individual characters and how they are related. For example, the Unicode database records whether a character is printable, is a letter, is a decimal digit, or is some other numeric symbol. That's how the `str` methods `isidentifier`, `isprintable`, `isdecimal`, and `isnumeric` work. `str.casefold` also uses information from a Unicode table.

## Finding characters by name

The `unicodedata` module has functions to retrieve character metadata, including `unicodedata.name()`, which returns a character's official name in the standard. [Figure 4-5](#) demonstrates that function<sup>13</sup>.

```
>>> from unicodedata import name
>>> name('A')
'LATIN CAPITAL LETTER A'
>>> name('ã')
'LATIN SMALL LETTER A WITH TILDE'
>>> name('♚')
'BLACK CHESS QUEEN'
>>> name('😺')
'GRINNING CAT FACE WITH SMILING EYES'
```

*Figure 4-5. Exploring unicodedata.name() in the Python console*

You can use the `name()` function to build apps that let users search for characters by name. [Figure 4-6](#) demonstrates the `cf.py` command-line script that takes one or more words as arguments, and lists the characters that have those words in their official Unicode names. The full source code for `cf.py` is in [Example 4-21](#).

```
$ ./cf.py cat smiling
U+1F638 😺 GRINNING CAT FACE WITH SMILING EYES
U+1F63A 😺 SMILING CAT FACE WITH OPEN MOUTH
U+1F63B 😺 SMILING CAT FACE WITH HEART-SHAPED EYES
(3 found)
```

*Figure 4-6. Using cf.py to find smiling cats.*

## WARNING

Emoji support varies widely accross desktop operating systems, shells, and apps. In recent years the MacOS terminal offers the best support for emojis, followed by modern GNU/Linux graphic terminals. Windows cmd.exe and PowerShell support Unicode output since 2018, but as I write this in January 2020, they still don't display emojis—at least not “out of the box”.

In [Example 4-21](#), note the `if` statement in the `find` function using the `.issubset()` method to quickly test whether all the words in the query set appear in the list of words built from the character's name. Thanks to Python's rich set API, we don't need a nested `for` loop and another `if` to implement this test.

*Example 4-21. cf.py: the character finder utility*

---

```
#!/usr/bin/env python3
import sys
import unicodedata

FIRST, LAST = ord(' '), sys.maxunicode ❶

def find(*query_words, first=FIRST, last=LAST): ❷
    query = {w.upper() for w in query_words} ❸
    count = 0
    for code in range(first, last + 1):
        char = chr(code) ❹
        name = unicodedata.name(char, None) ❺
        if name and query.issubset(name.split()): ❻
            print(f'U+{code:04X}\t{char}\t{name}') ❼
            count += 1
    print(f'({count} found)')

def main(words):
    if words:
        find(*words)
    else:
        print('Please provide words to find.')

if __name__ == '__main__':
    main(sys.argv[1:])
```

- ❶ Set defaults for first and last code points to search.
- ❷ `find` takes zero or more `query_words`, and optional keyword-only arguments to limit the range of the search, for easier testing.



- ③ Convert the `query_words` into a set of uppercased strings.
- ④ Get Unicode character for the code.
- ⑤ Get name of character, or `None` if the code point is unassigned.
- ⑥ If there is a name, split it into a list words, then check that `query` is a subset of that.
- ⑦ Print out line with code point in `U+9999` format, the character and its name.

The `unicodedata` module has other interesting functions. Next we'll see a few that are related to getting information from characters that have numeric meaning.

## Numeric meaning of characters

The `unicodedata` module includes functions to check whether a Unicode character represents a number and, if so, its numeric value for humans—as opposed to its code point number. [Example 4-22](#) shows the use of `unicodedata.name()` and `unicodedata.numeric()` along with the `.isdecimal()` and `.isnumeric()` methods of `str`.

*Example 4-22. Demo of Unicode database numerical character metadata (callouts describe each column in the output)*

---

```
import unicodedata
import re

re_digit = re.compile(r'\d')

sample = '1\u00bc\u00b2\u0969\u136b\u216b\u2466\u2480\u3285'

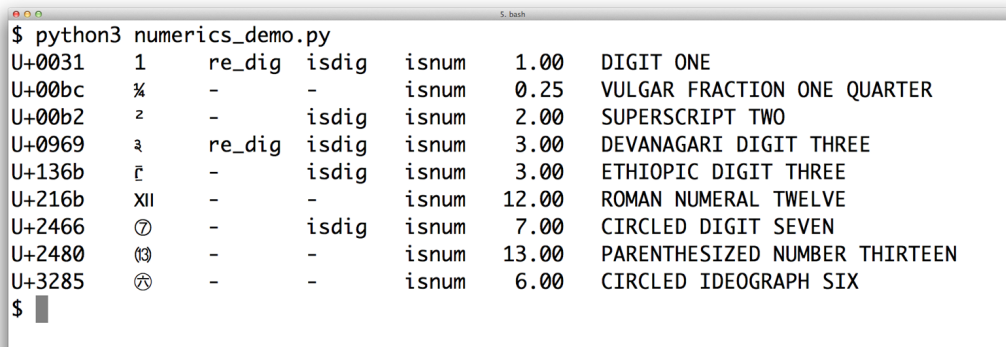
for char in sample:
    print('U+%04x' % ord(char),           ①
          char.center(6),                 ②
          're_digit' if re_digit.match(char) else '-', ③
          'isdigit' if char.isdigit() else '-',        ④
          'isnum' if char.isnumeric() else '-',        ⑤
          format(unicodedata.numeric(char), '5.2f'),   ⑥
```

```
unicodedata.name(char),  
sep='\t')
```

7

- ❶ Code point in U+0000 format.
- ❷ Character centralized in a str of length 6.
- ❸ Show `re_dig` if character matches the `r'\d'` regex.
- ❹ Show `isdig` if `char.isdigit()` is True.
- ❺ Show `isnum` if `char.isnumeric()` is True.
- ❻ Numeric value formatted with width 5 and 2 decimal places.
- ❼ Unicode character name.

Running Example 4-22 gives you Figure 4-7, if your terminal font has all those glyphs.



Code Point	Character	re_dig	isdig	isnum	Value	Name
U+0031	1	True	True	True	1.00	DIGIT ONE
U+00bc	¼	-	-	True	0.25	VULGAR FRACTION ONE QUARTER
U+00b2	²	-	True	True	2.00	SUPERSCRIPT TWO
U+0969	३	True	True	True	3.00	DEVANAGARI DIGIT THREE
U+136b	፫	-	True	True	3.00	ETHIOPIIC DIGIT THREE
U+216b	XII	-	-	True	12.00	ROMAN NUMERAL TWELVE
U+2466	⑦	-	True	True	7.00	CIRCLED DIGIT SEVEN
U+2480	⑬	-	-	True	13.00	PARENTHE SIZED NUMBER THIRTEEN
U+3285	⑆	-	-	True	6.00	CIRCLED IDEOGRAPH SIX

Figure 4-7. MacOS terminal showing numeric characters and metadata about them; `re_dig` means the character matches the regular expression `r'\d'`

The sixth column of Figure 4-7 is the result of calling `unicodedata.numeric(char)` on the character. It shows that Unicode knows the numeric value of symbols that represent numbers. So if you want to create a spreadsheet application that supports Tamil digits or Roman numerals, go for it!

Figure 4-7 shows that the regular expression `r'\d'` matches the digit “1” and the Devanagari digit 3, but not some other characters that are considered digits by the `isdigit` function. The `re` module is not as savvy about Unicode as it could be. The new `regex` module available in PyPI was designed to eventually replace `re` and provides better

Unicode support.<sup>14</sup> We'll come back to the `re` module in the next section.

Throughout this chapter we've used several `unicodedata` functions, but there are many more we did not cover. See the standard library documentation for the `unicodedata` module.

Next we'll take a quick look at a new trend: dual-mode APIs offering functions that accept `str` or `bytes` arguments with special handling depending on the type.

## Dual-Mode `str` and `bytes` APIs

Python's standard library has functions that accept `str` or `bytes` arguments and behave differently depending on the type. Some examples are in the `re` and `os` modules.

### `str` Versus `bytes` in Regular Expressions

If you build a regular expression with `bytes`, patterns such as `\d` and `\w` only match ASCII characters; in contrast, if these patterns are given as `str`, they match Unicode digits or letters beyond ASCII. [Example 4-23](#) and [Figure 4-8](#) compare how letters, ASCII digits, superscripts, and Tamil digits are matched by `str` and `bytes` patterns.

*Example 4-23. `ramanujan.py`: compare behavior of simple `str` and `bytes` regular expressions*

---

```
import re
```

```
re_numbers_str = re.compile(r'\d+')      ❶
re_words_str = re.compile(r'\w+')
re_numbers_bytes = re.compile(rb'\d+')    ❷
re_words_bytes = re.compile(rb'\w+')

text_str = ("Ramanujan saw \u0be7\u0bed\u0be8\u0bef"  ❸
            " as 1729 = 13 + 123 = 93 + 103." )  ❹
```

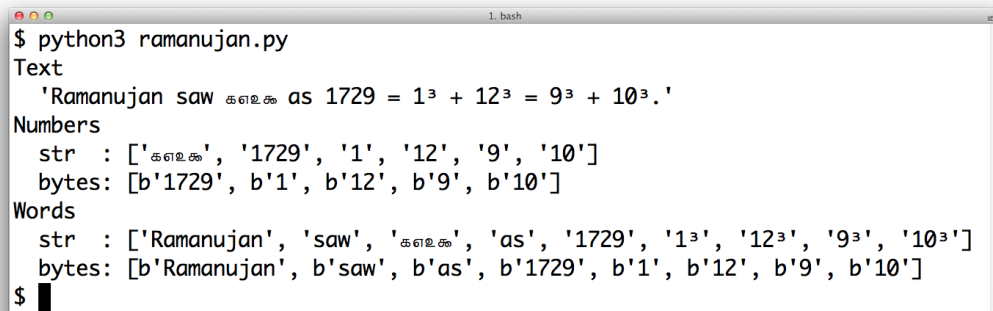
```

text_bytes = text_str.encode('utf_8') ⑤

print('Text', repr(text_str), sep='\n ')
print('Numbers')
print('  str  :', re_numbers_str.findall(text_str)) ⑥
print('  bytes:', re_numbers_bytes.findall(text_bytes)) ⑦
print('Words')
print('  str  :', re_words_str.findall(text_str)) ⑧
print('  bytes:', re_words_bytes.findall(text_bytes)) ⑨

```

- ❶ The first two regular expressions are of the `str` type.
- ❷ The last two are of the `bytes` type.
- ❸ Unicode text to search, containing the Tamil digits for 1729 (the logical line continues until the right parenthesis token).
- ❹ This string is joined to the previous one at compile time (see “2.4.2. String literal concatenation” in *The Python Language Reference*).
- ❺ A `bytes` string is needed to search with the `bytes` regular expressions.
- ❻ The `str` pattern `r'\d+'` matches the Tamil and ASCII digits.
- ❼ The `bytes` pattern `rb'\d+'` matches only the ASCII bytes for digits.
- ❽ The `str` pattern `r'\w+'` matches the letters, superscripts, Tamil, and ASCII digits.
- ❾ The `bytes` pattern `rb'\w+'` matches only the ASCII bytes for letters and digits.



```
$ python3 ramanujan.py
Text
'Ramanujan saw കണ്ടുക as 1729 = 1³ + 12³ = 9³ + 10³.'
Numbers
str : ['കണ്ടുക', '1729', '1', '12', '9', '10']
bytes: [b'1729', b'1', b'12', b'9', b'10']
Words
str : ['Ramanujan', 'saw', 'കണ്ടുക', 'as', '1729', '1³', '12³', '9³', '10³']
bytes: [b'Ramanujan', b'saw', b'as', b'1729', b'1', b'12', b'9', b'10']
$
```

Figure 4-8. Screenshot of running `ramanujan.py` from [Example 4-23](#)

[Example 4-23](#) is a trivial example to make one point: you can use regular expressions on `str` and `bytes`, but in the second case `bytes` outside of the ASCII range are treated as nondigits and nonword characters.

For `str` regular expressions, there is a `re.ASCII` flag that makes `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s`, and `\S` perform ASCII-only matching. See the [documentation of the `re` module](#) for full details.

Another important dual-mode module is `os`.

## str Versus bytes in os Functions

The GNU/Linux kernel is not Unicode savvy, so in the real world you may find filenames made of byte sequences that are not valid in any sensible encoding scheme, and cannot be decoded to `str`. File servers with clients using a variety of OSes are particularly prone to this problem.

In order to work around this issue, all `os` module functions that accept filenames or pathnames take arguments as `str` or `bytes`. If one such function is called with a `str` argument, the argument will be automatically converted using the codec named by `sys.getfilesystemencoding()`, and the OS response will be decoded with the same codec. This is almost always what you want, in keeping with the Unicode sandwich best practice.

But if you must deal with (and perhaps fix) filenames that cannot be handled in that way, you can pass `bytes` arguments to the `os` functions to get `bytes` return values. This feature lets you deal with any file or pathname, no matter how many gremlins you may find. See [Example 4-24](#).

*Example 4-24. `listdir` with `str` and `bytes` arguments and results*

---

```
>>> os.listdir('.') ❶  
['abc.txt', 'digits-of-π.txt']  
>>> os.listdir(b'.') ❷  
[b'abc.txt', b'digits-of-\\xcf\\x80.txt']
```

- ❶ The second filename is “digits-of-π.txt” (with the Greek letter pi).
- ❷ Given a byte argument, `listdir` returns filenames as bytes: `b'\\xcf\\x80'` is the UTF-8 encoding of the Greek letter pi).

To help with manual handling of `str` or `bytes` sequences that are file or pathnames, the `os` module provides special encoding and decoding functions `fsencode(name_or_path)` and `os.fsdecode(name_or_path)`. Both of these functions accept an argument of type `str`, `bytes`, or—since Python 3.6—an object implementing the `os.PathLike` interface.

Enough suffering. Let’s wrap up our tour of `str` versus `bytes` with a fun topic: building emojis.

## Multi-character emojis

As we saw in “[Normalizing Unicode for Saner Comparisons](#)”, it’s always been possible to produce accented characters by combining Unicode letters and diacritics. To accomodate the growing demand for emojis, this idea has been extended to produce different pictographs by combining special markers and emoji characters. Let’s start with the simplest kind of combined emoji: flags of countries.

## Country flags


Throughout history, countries split, join, mutate or simply adopt new flags. The Unicode consortium found a way to avoid keeping up with those changes and outsource the problem to the systems that claim Unicode support: its character database has no country flags. Instead there is a set of 26 “regional indicator symbols letters”, from A (U+1F1E6) to Z (U+1F1FF). When you combine two of those indicator letters to form an ISO 3166-1 country code, you get the corresponding country flag—if the UI supports it. [Example 4-25](#) shows how.

*Example 4-25. two\_flags.py: combining regional indicators to produce flags*

```
# REGIONAL INDICATOR SYMBOLS
RIS_A = '\U0001F1E6' # LETTER A
RIS_U = '\U0001F1FA' # LETTER U
print(RIS_A + RIS_U) # AU: Australia
print(RIS_U + RIS_A) # UA: Ukraine
print(RIS_A + RIS_A) # AA: no such country
```

Figure 4-9 shows the output of [Example 4-25](#) on a MacOS 10.14 terminal.

```
$ python3 two_flags.py
```



The screenshot shows the output of the program. The first line is the Australian flag (AU), the second is the Ukrainian flag (UA), and the third is a placeholder for the AA combination, which is displayed as two 'A' characters inside dashed squares.

*Figure 4-9. Screenshot of running two\_flags.py from Example 4-25. The AA combination is shown as two letters A inside dashed squares.*

If your program outputs a combination of indicator letters that is not recognized by the app, you get the indicators displayed as letters

inside dashed squares—again, depending on the UI. See the last line in [Figure 4-9](#).

## NOTE

Europe and the United Nations are not countries, but their flags are supported by the regional indicator pairs EU and UN, respectively. England, Scotland, and Wales may or may not be separate countries by the time you read this, but they also have flags supported by Unicode. However, instead of regional indicator letters, those flags require a more complicated scheme. Read [Emoji Flags Explained on Emojipedia](#) to learn how that works.

Now let's see how emoji modifiers can be used to set the skin tone of emojis that show human faces, hands, noses etc.

## Skin tones

Unicode provides a set of 5 emoji modifiers to set skin tone from pale to dark brown. They are based on the [Fitzpatrick scale](#)—developed to study the effects of ultraviolet light on human skin. [Example 4-26](#) shows the use of those modifiers to set the skin tone of the thumbs up emoji.

*Example 4-26. skin.py: the thumbs up emoji by itself, followed by all available skin tone modifiers.*

```
from unicodedata import name
```

```
SKIN1 = 0x1F3FB # EMOJI MODIFIER FITZPATRICK TYPE-1-2 ❶  
SKINS = [chr(i) for i in range(SKIN1, SKIN1 + 5)] ❷  
THUMB = '\U0001F44d' # THUMBS UP SIGN 👍
```

```
examples = [THUMB] ❸  
examples.extend(THUMB + skin for skin in SKINS) ❹
```



```

for example in examples:
    print(example, end='\t')
    print(' + '.join(name(char) for char in example))

```

- ❶ EMOJI MODIFIER FITZPATRICK TYPE-1-2 is the first modifier.
- ❷ Build list with all five modifiers.
- ❸ Start list with the unmodified THUMBS UP SIGN.
- ❹ Extend list with the same emoji followed by each of the modifiers.
- ❺ Display emoji and tab.
- ❻ Display names of characters combined in the emoji, joined by ' + '.

The output of [Example 4-26](#) looks like [Figure 4-10](#) on MacOS. As you can see, the unmodified emoji has a cartoonish yellow color, while the others have more realistic skin tones.

```

$ python3 skin.py
👍 THUMBS UP SIGN
👍 THUMBS UP SIGN + EMOJI MODIFIER FITZPATRICK TYPE-1-2
👍 THUMBS UP SIGN + EMOJI MODIFIER FITZPATRICK TYPE-3
👍 THUMBS UP SIGN + EMOJI MODIFIER FITZPATRICK TYPE-4
👍 THUMBS UP SIGN + EMOJI MODIFIER FITZPATRICK TYPE-5
👍 THUMBS UP SIGN + EMOJI MODIFIER FITZPATRICK TYPE-6

```

*Figure 4-10. Screenshot of [Example 4-26](#) in the MacOS 10.14 terminal.*

Let's now move to more complex emoji combinations using special markers.

## Rainbow flag and other ZWJ sequences

Besides the special purpose indicators and modifiers we've seen, Unicode provides a marker that is used as glue between emojis and other characters, to produce new combinations: U+200D, ZERO WIDTH JOINER—a.k.a. ZWJ in many Unicode documents.

For example rainbow flag is built by joining the emojis WAVING WHITE FLAG and RAINBOW, as Figure 4-11 shows.

```
>>> white_flag = '🚩' # U+1F3F3 WAVING WHITE FLAG
>>> vs16 = '\uFE0F' # VARIATION SELECTOR-16
>>> zwj = '\u200D' # ZERO WIDTH JOINER
>>> rainbow = '🌈' # U+1F308 RAINBOW
>>> print(white_flag + vs16 + zwj + rainbow)
```




Figure 4-11. Making the rainbow flag in the Python console.

Unicode 13 supports more than 1100 ZWJ emoji sequences as RGI —“recommended for general interchange [...] intended to be widely supported across multiple platforms”.<sup>15</sup> You can find the full list of RGI ZWJ emoji sequences in [emoji-zwj-sequences.txt](#) and a small sample in Figure 4-12.

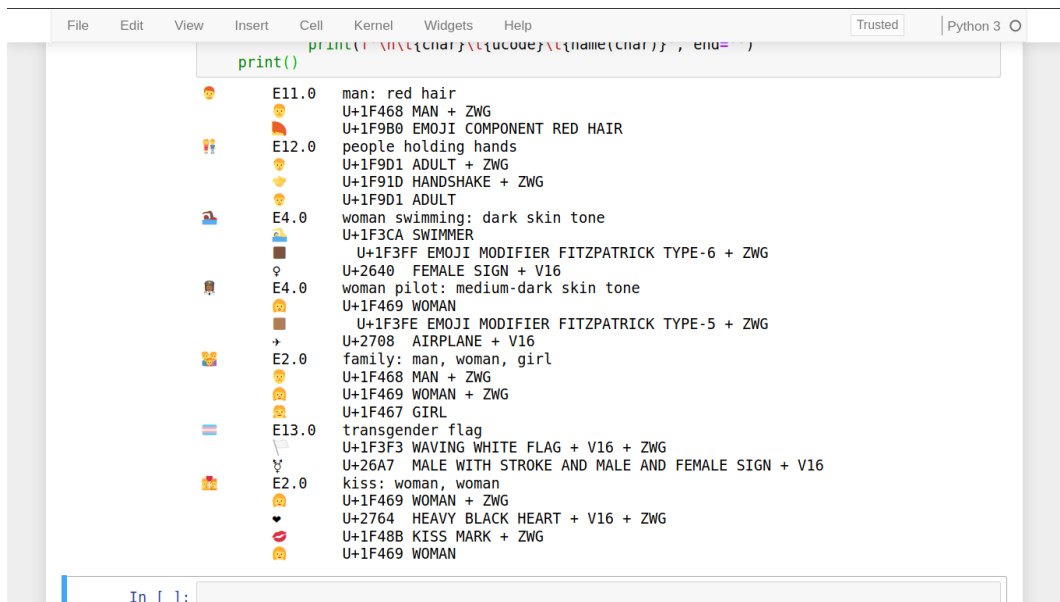


Figure 4-12. Sample ZWJ sequences generated by Example 4-27, running in a Jupyter Notebook, viewed on Firefox 72 on Ubuntu 19.10. This browser/OS combo can display all the emojis from this sample, including the newest: “people holding hands” and “transgender flag”, added in Emoji 12.0 and 13.0.

Example 4-27 is the source code that produced Figure 4-12. You can run it from your shell, but for better results I recommend pasting it

inside a Jupyter Notebook to run it in a browser. Browsers often lead the way in Unicode support, and provide prettier emoji pictographs.

*Example 4-27. zwj\_sample.py: produce listing with a few ZWJ characters.*

---

```
from unicodedata import name

zwg_sample = """
1F468 200D 1F9B0          |man: red hair
|E11.0
1F9D1 200D 1F91D 200D 1F9D1 |people holding hands
|E12.0
1F3CA 1F3FF 200D 2640 FE0F  |woman swimming: dark skin tone
|E4.0
1F469 1F3FE 200D 2708 FE0F  |woman pilot: medium-dark skin tone
|E4.0
1F468 200D 1F469 200D 1F467 |family: man, woman, girl
|E2.0
1F3F3 FE0F 200D 26A7 FE0F   |transgender flag
|E13.0
1F469 200D 2764 FE0F 200D 1F48B 200D 1F469 |kiss: woman, woman
|E2.0
"""

markers = {'\u200D': 'ZWJ', # ZERO WIDTH JOINER
           '\uFE0F': 'V16', # VARIATION SELECTOR-16
           }

for line in zwg_sample.strip().split('\n'):
    code, descr, version = (s.strip() for s in line.split('|'))
    chars = [chr(int(c, 16)) for c in code.split()]
    print(''.join(chars), version, descr, sep='\t', end='')
    while chars:
        char = chars.pop(0)
        if char in markers:
            print(' + ' + markers[char], end='')
        else:
            ucode = f'U+{ord(char):04X}'
            print(f'\n\t{char}\t{ucode}\t{name(char)}', end='')
    print()
```

One trend in modern Unicode is the addition of gender-neutral emojis such as SWIMMER (U+1F3CA) or ADULT (U+1F9D1), which can then be shown as they are, or with different gender in ZWJ sequences with the female sign ♀ (U+2640) or the male sign ♂ (U+2642). The Unicode Consortium is also moving towards more diversity in the supported family emojis. Figure 4-13 is a matrix of family emojis showing current support for families with different combinations of parents and children—as of January, 2020.

	👤	👩	👤+👤	👤+👩	👩+👤	👩+👩
👦	👤👩👦	👩👦	👤👤👦	👤👩👦	👩👤👦	👩👩👦
👧	👤👩👧	👩👧	👤👤👧	👤👩👧	👩👤👧	👩👩👧
👤+👤	👤👤👤	👤👤👩	👤👤👤👤	👤👤👤👩	👤👤👤👤	👤👤👤👩
👤+👩	👤👤👩	👤👤👩👩	👤👤👤👩	👤👤👤👩	👤👤👤👩	👤👤👤👩
👩+👤	👤👩👤	👤👩👤👩	👤👤👤👩	👤👤👤👩	👤👤👤👩	👤👤👤👩
👩+👩	👤👩👩	👤👩👩👩	👤👤👤👩	👤👤👤👩	👤👤👤👩	👤👤👤👩

Figure 4-13. The table shows adult singles and couples at the top, and boys and girls on the left side. Cells have the combined emoji of a family with the parent(s) from the top and kid(s) from the left. If a combination is not supported by the browser, more than one emoji will appear inside a cell. Firefox 72 on Windows 10 is able to show all combinations.

The code I wrote to build Figure 4-13 is mostly concerned with HTML formatting, but is listed in [Link to Come] for completeness.

Example 4-28.

Browsers follow the evolution of Unicode Emoji closely, and here no OS has a clear advantage. While preparing this chapter, I captured [Figure 4-12](#) on Ubuntu 19.10 and [Figure 4-13](#) on Windows 10, using Firefox 72 on both, because those were the OS/browser combinations with the most complete support for the emojis in those examples.

Unicode is a fascinating topic. However, now is time to wrap up our exploration of `str` and `bytes`.

## Chapter Summary

We started the chapter by dismissing the notion that `1 character == 1 byte`. As the world adopts Unicode, we need to keep the concept of text strings separated from the binary sequences that represent them in files, and Python 3 enforces this separation.

After a brief overview of the binary sequence data types—`bytes`, `bytearray`, and `memoryview`—we jumped into encoding and decoding, with a sampling of important codecs, followed by approaches to prevent or deal with the infamous `UnicodeEncodeError`, `UnicodeDecodeError`, and the `SyntaxError` caused by wrong encoding in Python source files.

While on the subject of source code, I presented my opinion on the debate about non-ASCII identifiers: if the maintainers of the code base want to use a human language that is not limited to ASCII characters, the identifiers should be spelled correctly. That’s precisely why Python 3 accepts non-ASCII identifiers.

We then considered the theory and practice of encoding detection in the absence of metadata: in theory, it can’t be done, but in practice the `Chardet` package pulls it off pretty well for a number of popular encodings. Byte order marks were then presented as the only encoding hint commonly found in UTF-16 and UTF-32 files—sometimes in UTF-8 files as well.

In the next section, we demonstrated opening text files, an easy task except for one pitfall: the `encoding=` keyword argument is not mandatory when you open a text file, but it should be. If you fail to specify the encoding, you end up with a program that manages to generate “plain text” that is incompatible across platforms, due to conflicting default encodings. We then exposed the different encoding settings that Python uses as defaults and how to detect them: `locale.getpreferredencoding()`,

`sys.getfilesystemencoding()`, `sys.getdefaultencoding()`, and the encodings for the standard I/O files (e.g., `sys.stdout.encoding`). A sad realization for Windows users is that these settings often have distinct values within the same machine, and the values are mutually incompatible; GNU/Linux and OSX users, in contrast, live in a happier place where UTF-8 is the default pretty much everywhere.

Text comparisons are surprisingly complicated because Unicode provides multiple ways of representing some characters, so normalizing is a prerequisite to text matching. In addition to explaining normalization and case folding, we presented some utility functions that you may adapt to your needs, including drastic transformations like removing all accents. We then saw how to sort Unicode text correctly by leveraging the standard `locale` module—with some caveats—and an alternative that does not depend on tricky locale configurations: the external PyUCA package.

Then we leveraged the Unicode database to build a command-line utility to search for characters by name—in 28 lines of code, thanks to the power of Python. We glanced at other Unicode metadata, and had a brief overview of dual-mode APIs (e.g., the `re` and `os` modules, where some functions can be called with `str` or `bytes` arguments, prompting different yet fitting results).

Finally, we saw how to produce flags, hands with different skin tones, family icons and other emoji combinations supported by Unicode.

## Further Reading

Ned Batchelder’s 2012 PyCon US talk “Pragmatic Unicode — or — How Do I Stop the Pain?” was outstanding. Ned is so professional that he provides a full transcript of the talk along with the slides and video. Esther Nam and Travis Fischer gave an excellent PyCon 2014 talk “Character encoding and Unicode in Python: How to (◡ ◡◡) ◡

L with dignity” (slides, video), from which I quoted this chapter’s short and sweet epigraph: “Humans use text. Computers speak bytes.” Lennart Regebro—one of this book’s technical reviewers—presents his “Useful Mental Model of Unicode (UMMU)” in the short post [“Unconfusing Unicode: What Is Unicode?”](#). Unicode is a complex standard, so Lennart’s UMMU is a really useful starting point.

The official [Unicode HOWTO](#) in the Python docs approaches the subject from several different angles, from a good historic intro to syntax details, codecs, regular expressions, filenames, and best practices for Unicode-aware I/O (i.e., the Unicode sandwich), with plenty of additional reference links from each section. [Chapter 4, “Strings”](#), of Mark Pilgrim’s awesome book *Dive into Python 3* also provides a very good intro to Unicode support in Python 3. In the same book, [Chapter 15](#) describes how the Chardet library was ported from Python 2 to Python 3, a valuable case study given that the switch from the old `str` to the new `bytes` is the cause of most migration pains, and that is a central concern in a library designed to detect encodings.

If you know Python 2 but are new to Python 3, Guido van Rossum’s [What’s New in Python 3.0](#) has 15 bullet points that summarize what changed, with lots of links. Guido starts with the blunt statement: “Everything you thought you knew about binary data and Unicode has changed.” Armin Ronacher’s blog post [“The Updated Guide to Unicode on Python”](#) is deep and highlights some of the pitfalls of Unicode in Python 3 (Armin is not a big fan of Python 3).

Chapter 2, “Strings and Text,” of the [Python Cookbook, Third Edition](#) (O’Reilly), by David Beazley and Brian K. Jones, has several recipes dealing with Unicode normalization, sanitizing text, and performing text-oriented operations on byte sequences. Chapter 5 covers files and I/O, and it includes [“Recipe 5.17. Writing Bytes to a Text File,”](#) showing that underlying any text file there is always a binary stream



that may be accessed directly when needed. Later in the cookbook, the `struct` module is put to use in “Recipe 6.11. Reading and Writing Binary Arrays of Structures.”

Nick Coghlan’s Python Notes blog has two posts very relevant to this chapter: “[Python 3 and ASCII Compatible Binary Protocols](#)” and “[Processing Text Files in Python 3](#)”. Highly recommended.

A list of encodings supported by Python is available at [Standard Encodings](#) in the `codecs` module documentation. If you need to get that list programmatically, see how it’s done in the [/Tools/unicode/listcodecs.py](#) script that comes with the CPython source code.

Martijn Faassen’s “[Changing the Python Default Encoding Considered Harmful](#)” and Tarek Ziade’s “[sys.setdefaultencoding Is Evil](#)” explain why the default encoding you get from `sys.getdefaultencoding()` should never be changed, even if you discover how.

The books [Unicode Explained](#) by Jukka K. Korpela (O’Reilly) and [Unicode Demystified](#) by Richard Gillam (Addison-Wesley) are not Python-specific but were very helpful as I studied Unicode concepts. [Programming with Unicode](#) by Victor Stinner is a free, self-published book (Creative Commons BY-SA) covering Unicode in general as well as tools and APIs in the context of the main operating systems and a few programming languages, including Python.

The W3C pages [Case Folding: An Introduction](#) and [Character Model for the World Wide Web: String Matching and Searching](#) cover normalization concepts, with the former being a gentle introduction and the latter a working group note written in dry standard-speak—the same tone of the [Unicode Standard Annex #15 — Unicode Normalization Forms](#). The [Frequently Asked Questions / Normalization](#) from [Unicode.org](#) is more readable, as is the [NFC](#)

[FAQ](#) by Mark Davis—author of several Unicode algorithms and president of the Unicode Consortium at the time of this writing. To learn more about Unicode Emoji standards, visit the [Unicode Emoji index](#) page, which links to the [Technical Standard #51: Unicode Emoji](#) and the emoji data files, where you'll find [emoji-zwj-sequences.txt](#)—the source of the samples I used in [Figure 4-12](#).

[Emojipedia](#) is the best site to find emojis and learn about them. Besides a comprehensive searchable database, Emojipedia also has a blog including posts like [Emoji ZWJ Sequences: Three Letters, Many Possibilities](#) and [Emoji Flags Explained](#).

In 2016, the Museum of Modern Art (MoMA) in NYC added to its collection [The Original Emoji](#), the 176 emojis designed by Shigetaka Kurita in 1999 for NTT DOCOMO—the Japanese mobile carrier. Going further back in history, Emojipedia published [Correcting the Record on the First Emoji Set](#), crediting Japan's SoftBank for the earliest known emoji set, deployed in cell phones in 1997. SoftBank's set is the source of 90 emojis now in Unicode, including U+1F4A9 (PILE OF POO). The culture and politics of emoji evolution in the 2010-2019 decade are the subject of Paddy Johnson's article [Emoji We Lost](#) for Gizmodo. Matthew Rothenberg's [emojitracker.com](#) is a live dashboard showing counts of emoji usage on Twitter, updated in real time. As I write this, FACE WITH TEARS OF JOY (U+1F602) is the most popular emoji on Twitter, with 2,693,102,686 recorded occurrences.

## SOAPBOX

### What Is “Plain Text”?

For anyone who deals with non-English text on a daily basis, “plain text” does not imply “ASCII.” The Unicode Glossary defines *plain text* like this:

*Computer-encoded text that consists only of a sequence of code points from a given standard, with no other formatting or structural information.*

That definition starts very well, but I don’t agree with the part after the comma. HTML is a great example of a plain-text format that carries formatting and structural information. But it’s still plain text because every byte in such a file is there to represent a text character, usually using UTF-8. There are no bytes with nontext meaning, as you can find in a *.png* or *.xls* document where most bytes represent packed binary values like RGB values and floating-point numbers. In plain text, numbers are represented as sequences of digit characters.

I am writing this book in a plain-text format called—ironically—AsciiDoc, which is part of the toolchain of O’Reilly’s excellent Atlas book publishing platform. AsciiDoc source files are plain text, but they are UTF-8, not ASCII. Otherwise, writing this chapter would have been really painful. Despite the name, AsciiDoc is just great.

The world of Unicode is constantly expanding and, at the edges, tool support is not always there. Not all characters I wanted to show were available in the fonts used to render the book. That’s why I had to use images for instead of listings in several examples in this chapter. On the other hand, the Ubuntu and MacOS terminals display most Unicode text very well—including the Japanese characters for the word “mojibake”: 文字化け.

### Unicode Riddles

Imprecise qualifiers such as “often,” “most,” and “usually” seem to pop up whenever I write about Unicode normalization. I regret the lack of more definitive advice, but there are so many exceptions to the rules in Unicode that it is hard to be absolutely positive.

For example, the  $\mu$  (micro sign) is considered a “compatibility character” but the  $\Omega$  (ohm) and Å (Ångström) symbols are not. The difference has practical consequences: NFC normalization—recommended for text matching—replaces the  $\Omega$  (ohm) by  $\Omega$  (uppercase Greek omega) and the Å (Ångström) by Å (uppercase A with ring above). But as a “compatibility character” the  $\mu$  (micro sign) is not replaced by the visually identical  $\mu$  (lowercase Greek mu), except when the stronger NFKC or NFKD normalizations are applied, and these transformations are lossy.

I understand the  $\mu$  (micro sign) is in Unicode because it appears in the `latin1` encoding and replacing it with the Greek mu would break round-trip conversion. After all, that's why the micro sign is a "compatibility character." But if the ohm and Ångström symbols are not in Unicode for compatibility reasons, then why have them at all? There are already code points for the GREEK CAPITAL LETTER OMEGA and the LATIN CAPITAL LETTER A WITH RING ABOVE, which look the same and replace them on NFC normalization. Go figure.

My take after many hours studying Unicode: it is hugely complex and full of special cases, reflecting the wonderful variety of human languages and the politics of industry standards.

### **The power of soccer**

Here is another Unicode mystery: why are England, Scotland, and Wales entitled to their own Unicode flags, but Catalonia, Pernambuco, and Texas are not? Because the first three are permanent members of the International Football Association Board which controls the rules of soccer. As such, they are allowed to enter their "national teams" in the FIFA World Cup—therefore media outlets need their flags to display tournament charts. At least that's my theory.

### **How Are str Represented in RAM?**

The official Python docs avoid the issue of how the code points of a `str` are stored in memory. This is, after all, an implementation detail. In theory, it doesn't matter: whatever the internal representation, every `str` must be encoded to bytes on output.

In memory, Python 3 stores each `str` as a sequence of code points using a fixed number of bytes per code point, to allow efficient direct access to any character or slice.

Before Python 3.3, CPython could be compiled to use either 16 or 32 bits per code point in RAM; the former was a "narrow build," and the latter a "wide build." To know which you have, check the value of `sys.maxunicode`: 65535 implies a "narrow build" that can't handle code points above U+FFFF transparently. A "wide build" doesn't have this limitation, but used a lot of more memory: 4 bytes per character, even while the vast majority of code points for Chinese ideographs fit in 2 bytes. Neither option was great, so you had to choose depending on your needs.

Since Python 3.3, when creating a new `str` object, the interpreter checks the characters in it and chooses the most economic memory layout that is suitable for that particular `str`: if there are only characters in the `latin1` range, that `str` will use just one byte per code point. Otherwise, 2 or 4 bytes per code point

may be used, depending on the `str`. This is a simplification; for the full details, look up [PEP 393 — Flexible String Representation](#).

The flexible string representation is similar to the way the `int` type works in Python 3: if the integer fits in a machine word, it is stored in one machine word. Otherwise, the interpreter switches to a variable-length representation like that of the Python 2 `long` type. It is nice to see the spread of good ideas.

However, we can always count on Armin Ronacher to find problems in Python 3. He explained to me personally why that was not such a great idea in practice: it takes a single RAT (U+1F400) to inflate an otherwise all-ASCII text into a memory-hogging internal array using 4 bytes per character. In addition, because of all the ways Unicode characters combine, the ability to retrieve an arbitrary character by position is overrated—and extracting arbitrary slices from Unicode text is naïve at best, and often wrong. As emojis become more popular, these problems will only get worse.

- 
- [1](#) Slide 12 of PyCon 2014 talk “Character Encoding and Unicode in Python” ([slides](#), [video](#)).
  - [2](#) Python 2.6 and 2.7 also have `bytes`, but it’s just an alias to the `str` type, and does not behave like the Python 3 `bytes` type.
  - [3](#) It did not work in Python 3.0 to 3.4, causing much pain to developers dealing with binary data. The reversal is documented in [PEP 461 — Adding % formatting to bytes and bytearray](#).
  - [4](#) I first saw the term “Unicode sandwich” in Ned Batchelder’s excellent [“Pragmatic Unicode”](#) talk at US PyCon 2012.
  - [5](#) Python 2.6 or 2.7 users have to use `io.open()` to get automatic decoding/encoding when reading/writing.
  - [6](#) Source: [Windows Command-Line: Unicode and UTF-8 Output Text Buffer](#).
  - [7](#) While researching this subject, I did not find a list of situations when Python 3 internally converts `bytes` to `str`. Python core developer Antoine Pitrou says on the [comp.python.devel](#) list that CPython

internal functions that depend on such conversions “don’t get a lot of use in py3k.”

- 8 The Python 2 `sys.setdefaultencoding` function was misused and is no longer documented in Python 3. It was intended for use by the core developers when the internal default encoding of Python was still undecided. In the same [`comp.python.devel` thread](#), Marc-André Lemburg states that the `sys.setdefaultencoding` must never be called by user code and the only values supported by CPython are 'ascii' in Python 2 and 'utf-8' in Python 3.
- 9 Curiously, the micro sign is considered a “compatibility character” but the ohm symbol is not. The end result is that NFC doesn’t touch the micro sign but changes the ohm symbol to capital omega, while NFKC and NFKD change both the ohm and the micro into Greek characters.
- 10 Diacritics affect sorting only in the rare case when they are the only difference between two words—in that case, the word with a diacritic is sorted after the plain word.
- 11 Thanks to Leonardo Rochaël who went beyond his duties as tech reviewer and researched these Windows details, even though he is a GNU/Linux user himself.
- 12 Again, I could not find a solution, but did find other people reporting the same problem. Alex Martelli, one of the tech reviewers, had no problem using `setlocale` and `locale.strxfrm` on his Mac with OSX 10.9. In summary: your mileage may vary.
- 13 That’s an image—not a code listing—because emojis are not well supported by O’Reilly’s digital publishing toolchain as I write this.
- 14 Although it was not better than `re` at identifying digits in this particular sample.
- 15 Definition quoted from [`Technical Standard #51 Unicode Emoji`](#).

# Chapter 5. Record-like data structures

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at [fluentpython2e@ramalho.org](mailto:fluentpython2e@ramalho.org).

*Data classes are like children. They are okay as a starting point, but to participate as a grownup object, they need to take some responsibility.*<sup>1</sup>

—Martin Fowler and Kent Beck

Python offers a few ways to build a simple class that is just a bunch of fields, with little or no extra functionality. That pattern is known as a “data class”—and `dataclass` is the name of a Python decorator that supports it. This chapter covers three different class builders that you may use as shortcuts to write data classes:

- `collections.namedtuple`: the simplest way—since Python 2.6;
- `typing.NamedTuple`: an alternative that allows type annotations on the fields—since Python 3.5; `class` syntax supported since 3.6;
- `@dataclasses.dataclass`: a class decorator that allows more customization than previous alternatives, adding lots of

options and potential complexity—since Python 3.7.

After covering those class builders, we will discuss why *Data Class* is also the name of a code smell: a coding pattern that may be a symptom of poor object-oriented design.

The chapter ends with a section on a very different topic, but still closely related to record-like data: the `struct` module, designed to parse and build packed binary records that you may find in legacy flat-file databases, network protocols, and file headers.

### NOTE

`typing.TypedDict` (since Python 3.8) may seem like another data class builder—it’s described right after `typing.NamedTuple` in the [typing module documentation](#), and uses similar syntax. However, `TypedDict` does not build concrete classes that you can instantiate. It’s just a way to write static annotations for variables and function arguments that are expected to accept plain dictionaries with a fixed set of keys and a specific type for the value mapped to each key.

## What’s new in this chapter

This chapter is new in *Fluent Python 2<sup>nd</sup> edition*. The sections “[Classic Named Tuples](#)” and “[Structs and Memory Views](#)” appeared in chapters 2 and 4 in the *1<sup>st</sup> edition*, but the rest of the chapter is completely new.

We begin with a high level overview of the three class builders.

## Overview of data class builders

Consider a simple class to represent a geographic coordinate pair:



### Example 5-1. *class/coordinates.py*

---

**class** **Coordinate**:

```
def __init__(self, lat, long):
    self.lat = lat
    self.long = long
```

That `Coordinate` class does the job of holding latitude and longitude attributes. Writing the `__init__` boilerplate becomes old real fast, especially if your class has more than a couple of attributes: each of them is mentioned three times! And that boilerplate doesn't buy us basic features we'd expect from a Python object:

```
>>> from coordinates import Coordinate
>>> moscow = Coordinate(55.76, 37.62)
>>> moscow
<coordinates.Coordinate object at 0x107142f10> ❶
>>> location = Coordinate(55.76, 37.62)
>>> location == moscow ❷
False
>>> (location.lat, location.long) == (moscow.lat,
moscow.long) ❸
True
```

- ❶ `__repr__` inherited from `object` is not very helpful.
- ❷ Meaningless equality; the `__eq__` method inherited from `object` compares object ids.
- ❸ Comparing two coordinates requires explicit comparison of each attribute.

The data class builders covered in this chapter provide the necessary `__init__`, `__repr__`, and `__eq__` methods automatically, as well as other useful features.

## NOTE

None of the class builders discussed here depend on inheritance to do their work. Both `collections.namedtuple` and `typing.NamedTuple` build classes that are `tuple` subclasses. `@dataclass` is a class decorator that does not affect the class hierarchy in any way. Each of them use different metaprogramming techniques to inject methods and data attributes into the class under construction.

Here is a `Coordinate` class built with `namedtuple`—a factory function that builds a subclass of `tuple` with the name and fields you specify:

```
>>> from collections import namedtuple
>>> Coordinate = namedtuple('Coordinate', 'lat long')
>>> issubclass(Coordinate, tuple)
True
>>> moscow = Coordinate(55.756, 37.617)
>>> moscow
Coordinate(lat=55.756, long=37.617) ❶
>>> moscow == Coordinate(lat=55.756, long=37.617) ❷
True
```

- ❶ Useful `__repr__`.
- ❷ Meaningful `__eq__`.

The newer `typing.NamedTuple` provides the same functionality, adding a type annotation to each field:

```
>>> import typing
>>> Coordinate = typing.NamedTuple('Coordinate', [('lat', float), ('long', float)])
>>> issubclass(Coordinate, tuple)
True
```

```
>>> Coordinate.__annotations__  
{'lat': <class 'float'>, 'long': <class 'float'>}
```

### TIP

A typed named tuple can also be constructed with the fields given as keyword arguments, like this:

```
Coordinate = typing.NamedTuple('Coordinate', lat=float,  
                                long=float)
```

This is more readable, and also lets you provide the mapping of fields and types as `**fields_and_types`.

Since Python 3.6, `typing.NamedTuple` can also be used in a `class` statement, with type annotations written as described in [PEP 526—Syntax for Variable Annotations](#). This is much more readable, and makes it easy to override methods or add new ones. [Example 5-2](#) is the same `Coordinate` class, with a pair of `float` attributes and a custom `__str__` to display a coordinate formatted like 55.8°N, 37.6°E:

*Example 5-2. `typing_namedtuple/coordinates.py`*

---

```
from typing import NamedTuple
```

```
class Coordinate(NamedTuple):
```

```
    lat: float  
    long: float
```

```
    def __str__(self):  
        ns = 'N' if self.lat >= 0 else 'S'  
        we = 'E' if self.long >= 0 else 'W'
```

```
return f'{abs(self.lat):.1f}°{ns}, {abs(self.long):.1f}°  
{we}'
```

## WARNING

Although `NamedTuple` appears in the `class` statement as a superclass, it's actually not. `typing.NamedTuple` uses the advanced functionality of a metaclass<sup>2</sup> to customize the creation of the user's class. Check this out:

```
>>> issubclass(Coordinate, typing.NamedTuple)  
False  
>>> issubclass(Coordinate, tuple)  
True
```

In the `__init__` method generated by `typing.NamedTuple`, the fields appear as parameters in the same order they appear in the `class` statement.

Like `typing.NamedTuple`, the `dataclass` decorator supports [PEP 526](#) syntax to declare instance attributes. The decorator reads the variable annotations and automatically generates methods for your class. For comparison, check out the equivalent `Coordinate` class written with the help of the `dataclass` decorator:

*Example 5-3. `dataclass/coordinates.py`*

---

```
from dataclasses import dataclass
```

```
@dataclass(frozen=True)  
class Coordinate:
```

```
    lat: float  
    long: float
```

```
def __str__(self):
    ns = 'N' if self.lat >= 0 else 'S'
    we = 'E' if self.long >= 0 else 'W'
    return f'{abs(self.lat):.1f}°{ns}, {abs(self.long):.1f}°{we}'
```

Note that the body of the classes in [Example 5-2](#) and [Example 5-3](#) are identical—the difference is in the `class` statement itself. The `@dataclass` decorator does not depend on inheritance or a metaclass, so it should not interfere with your own use of these mechanisms.<sup>3</sup> The `Coordinate` class in [Example 5-3](#) is a subclass of `object`.

## Main features

The different data class builders have a lot of common. Here we'll discuss the main features they share. [Table 5-1](#) summarizes.

*Table 5-1. Selected features compared accross the three data class builders. *x* stands for an instance of a data class of that kind.*

	<b>namedtuple</b>	<b>NamedTuple</b>	<b>dataclass</b>
mutable instances	NO	NO	YES
class statement syntax	NO	YES	YES
construct dict	<code>x._asdict()</code>	<code>x._asdict()</code>	<code>dataclasses.as_dict(x)</code>
get field names	<code>x._fields</code>	<code>x._fields</code>	<code>[f.name for f in dataclasses.fields(x)]</code>
get defaults	<code>x._field_defaults</code>	<code>x._field_defaults</code>	<code>[f.default for f in dataclasses.fields(x)]</code>
get field types	N/A	<code>x.__annotations__</code>	<code>x.__annotations__</code>
new instance with changes	<code>x._replace(...)</code>	<code>x._replace(...)</code>	<code>dataclasses.replace(x, ...)</code>
new class at runtime	<code>namedtuple(...)</code>	<code>NamedTuple(...)</code>	<code>dataclasses.make_dataclass(...)</code>

## MUTABLE INSTANCES

A key difference between these class builders is that `collections.namedtuple` and `typing.NamedTuple` build tuple subclasses, therefore the instances are immutable. By default, `@dataclass` produces mutable classes. But the decorator accepts several keyword arguments to configure the class, including `frozen`—shown in [Example 5-3](#). When `frozen=True`, the class will raise an

exception if you try to assign values to the fields after the instance is initialized.

## CLASS STATEMENT SYNTAX

`typing.NamedTuple` and `dataclass` support the regular `class` statement syntax, making it easier to add methods and docstrings to the class you are creating; `collections.namedtuple` does not support that syntax.

## CONSTRUCT DICT

Both named tuple variants provide an instance method (`._as_dict`) to to construct a `dict` object from the fields in a data class instance. `dataclass` avoids injecting a similar method in the data class, but provides a module-level function to do it: `dataclasses.as_dict`.

## GET FIELD NAMES AND DEFAULT VALUES

All three class builders let you get the field names and default values that may be configured for them. In named tuple classes, that metadata is in the `._fields` and `._fields_defaults` class attributes. You can get the same metadata from a `dataclass` decorated class using the `fields` function from the `dataclasses` module. It returns a tuple of `Field` objects which have several attributes, including `name` and `default`.

## GET FIELD TYPES

A mapping of field names to type annotations is stored in the `__annotations__` class attribute in classes defined with the help of `typing.NamedTuple` and `dataclass`.

## NEW INSTANCE WITH CHANGES

Given a named tuple instance `x`, the call `x._replace(**kwargs)` returns a new instance with some attribute values replaced according

to the keyword arguments given. The `dataclasses.replace(x, **kwargs)` module-level function does the same for an instance of a `dataclass` decorated class.

## NEW CLASS AT RUNTIME

Although the `class` statement syntax is more readable, it is hard-coded. A framework may need to build data classes on the fly, at runtime. For that, you can use the default function call syntax of `collections.namedtuple`, which is likewise supported by `typing.NamedTuple`. The `dataclasses` module provides a `make_dataclass` function for the same purpose.

After this overview of the main features of the data class builders, let's focus on each of them in turn, starting with the simplest.

## Classic Named Tuples

The `collections.namedtuple` function is a factory that builds subclasses of `tuple` enhanced with field names, a class name, and a nice `__repr__`--which helps debugging. Classes built with `namedtuple` can be used anywhere where `tuples` are needed, and in fact many functions of the Python standard library that used to return `tuples` now return named tuples for convenience, without affecting user's code at all.

### TIP

Each instance of a class built by `namedtuple` takes exactly the same amount of memory a `tuple` because the field names are stored in the class. They use less memory than a regular object because they don't store attributes as key-value pairs in one `__dict__` for each instance.



Example 5-4 shows how we could define a named tuple to hold information about a city.

*Example 5-4. Defining and using a named tuple type*

---

```
>>> from collections import namedtuple
>>> City = namedtuple('City', 'name country population
coordinates') ❶
>>> tokyo = City('Tokyo', 'JP', 36.933, (35.689722, 139.691667))
❷
>>> tokyo
City(name='Tokyo', country='JP', population=36.933, coordinates=
(35.689722,
139.691667))
>>> tokyo.population ❸
36.933
>>> tokyo.coordinates
(35.689722, 139.691667)
>>> tokyo[1]
'JP'
```

- ❶ Two parameters are required to create a named tuple: a class name and a list of field names, which can be given as an iterable of strings or as a single space-delimited string.
- ❷ Field values must be passed as separate positional arguments to the constructor (in contrast, the `tuple` constructor takes a single iterable).
- ❸ You can access the fields by name or position.

As a `tuple` subclass, `City` inherits useful methods such as `__eq__` and the special methods for comparison operators (`__gt__`, `__ge__`, etc.) which are useful for sorting sequences of `City`.

In addition to the methods inherited from `tuple`, a named tuple offers a few attributes and methods. Example 5-5 shows the most useful: the `_fields` class attribute, the class method `_make(iterable)`, and the `_asdict()` instance method.

*Example 5-5. Named tuple attributes and methods (continued from the previous example)*

---

```
>>> City._fields ❶
('name', 'country', 'population', 'location')
>>> Coordinate = namedtuple('Coordinate', 'lat long')
>>> delhi_data = ('Delhi NCR', 'IN', 21.935,
Coordinate(28.613889, 77.208889))
>>> delhi = City._make(delhi_data) ❷
>>> delhi._asdict() ❸
{'name': 'Delhi NCR', 'country': 'IN', 'population': 21.935,
'location': Coordinate(lat=28.613889, long=77.208889)}
>>> import json
>>> json.dumps(delhi._asdict()) ❹
'{"name": "Delhi NCR", "country": "IN", "population": 21.935,
"location": [28.613889, 77.208889]}'
```

- ❶ `._fields` is a tuple with the field names of the class.
- ❷ `._make()` builds `City` from an iterable; `City(*delhi_data)` would do the same.
- ❸ `._asdict()` returns a `dict` built from the named tuple instance.
- ❹ `._asdict()` is useful to serialize the data in JSON format, for example.

### WARNING

The `._asdict` method returned an `OrderedDict` in Python 2.7, and in Python 3.1 to 3.7. Since Python 3.8, a regular `dict` is returned—which is probably fine now that we can rely on key insertion order. If you must have an `OrderedDict`, the [`.\_asdict` documentation](#) recommends building one from the result: `OrderedDict(x._asdict())`.

Since Python 3.7, `namedtuple` accepts the `defaults` keyword-only argument providing an iterable of `N` default values for each of the `N`

rightmost fields of the class. [Example 5-6](#) show how to define a `Coordinate` named tuple with a default value for a reference field:

*Example 5-6. Named tuple attributes and methods (continued from the previous example)*

---

```
>>> Coordinate = namedtuple('Coordinate', 'lat long reference',
defaults=['WGS84'])
>>> Coordinate(0, 0)
Coordinate(lat=0, long=0, reference='WGS84')
>>> Coordinate._field_defaults
{'reference': 'WGS84'}
```

In “[Class statement syntax](#)” I mentioned it’s easier to code methods with the class syntax supported by `typing.NamedTuple` and `@dataclass`. You can also add methods to a `namedtuple`, but it’s a hack. Skip the following box if you’re not interested in hacks.

## HACKING A NAMEDTUPLE TO INJECT A METHOD

Recall how we built the `Card` class in [Example 1-1](#) in [Chapter 1](#):

```
Card = collections.namedtuple('Card', ['rank', 'suit'])
```

Later [Chapter 1](#) I wrote a `spades_high` function for sorting. It would be nice if that logic was encapsulated in method of `Card`, but adding `spades_high` to `Card` without the benefit of a `class` statement requires a quick hack: define the function and then assign it to a class attribute. [Example 5-7](#) shows how.

*Example 5-7. frenchdeck.doctest: Adding a class attribute and a method to `Card`, the `namedtuple` from “A Pythonic Card Deck”*

```
>>> Card.suit_values = dict(spades=3, hearts=2, diamonds=1, clubs=0)
❶
>>> def spades_high(card):
❷
...     rank_value = FrenchDeck.ranks.index(card.rank)
...     suit_value = card.suit_values[card.suit]
...     return rank_value * len(card.suit_values) + suit_value
...
>>> Card.overall_rank = spades_high
❸
>>> lowest_card = Card('2', 'clubs')
>>> highest_card = Card('A', 'spades')
>>> lowest_card.overall_rank()
❹
0
>>> highest_card.overall_rank()
51
```

- ❶ Attach a class attribute with values for each suit.
- ❷ `spades_high` will become a method; the first argument doesn't need to be named `self`, but it must refer to the receiver (the target instance, which we usually call `self`).
- ❸ Attach the function to the `Cards` class. It becomes a method named `overall_rank`.
- ❹ It works!

For readability and future maintenance, it's much better to be able to code methods inside a `class` statement. But it's good to know this hack is possible, because it may come in handy.<sup>4</sup>

This was small detour to showcase the power of a dynamic language. Now, on to the next features of the data class builders.

Now let's check out the `typing.NamedTuple` variation.

## Typed Named Tuples

The `Coordinate` class with a default field from [Example 5-6](#) can be written like this using `typing.NamedTuple`:

*Example 5-8. `typing_namedtuple/coordinates2.py`*

```
from typing import NamedTuple
```

```
class Coordinate(NamedTuple):
```

```
    lat: float          ❶  
    long: float  
    reference: str = 'WGS84' ❷
```

- ❶ Every instance field must be annotated with a type.
- ❷ The reference instance field is annotated with a type and a default value

Classes built by `typing.NamedTuple` don't have any methods beyond those that `collections.namedtuple` also generates—and those that are inherited from `tuple`. The only difference at runtime is the presence of the `__attributes__` class field—which Python completely ignores at runtime.

## WARNING

Classes built with `typing.NamedTuple` also have a `_field_types` attribute. Since Python 3.8, that attribute is deprecated in favor of `__annotations__` which has the same information and is the canonical place to find type hints in Python objects that have them.

Given that the main feature of `typing.NamedTuple` are the type annotations, we'll take a brief look at them before resuming our exploration of data class builders.

## Type hints 101

Type hints—a.k.a. type annotations—are ways to declare the expected type of function arguments, return values, and variables.

## NOTE

This is a very brief introduction to type hints, just enough to make sense of the syntax and meaning of the annotations used `typing.NamedTuple` and `@dataclass` declarations. We will cover type hints for function signatures in [\[Link to Come\]](#) and more advanced annotations like generics, unions etc. in [\[Link to Come\]](#). Here we'll mostly see hints with built-in types, such as `str`, `int`, and `float`, which are probably the most common types used to annotate fields of data classes.

The first thing you need to know about type hints is that they are not enforced at all by the Python bytecode compiler and runtime interpreter.

## No runtime effect

Type annotations don't have any impact on the runtime behavior of Python programs. Check this out:

*Example 5-9. Python does not enforce type hints at runtime.*

```
>>> import typing
>>> class Coordinate(typing.NamedTuple):
...     lat: float
...     long: float
...
>>> trash = Coordinate('foo', None)
>>> print(trash)
Coordinate(lat='Ni!', long=None) ❶
```

❶ I told you: no type checking at runtime!

If you type the code of [Example 5-9](#) in a Python module, replacing the last line with `print(trash)`, it will happily run and display a meaningless `Coordinate`, with no error or warning:

```
$ python3 nocheck_demo.py
Coordinate(lat='Ni!', long=None)
```

The type hints are intended primarily to support third-party type checkers, like [Mypy](#) or the [PyCharm IDE](#) built-in type checker. These are static analysis tools: they check Python source code “at rest”, not running code.

To see the effect of type hints, you must run one of those tools on your code—like a linter. For instance, here is what [Mypy](#) has to say about the previous example:

```
$ mypy nocheck_demo.py
nocheck_demo.py:8: error: Argument 1 to "Coordinate" has
incompatible type "str"; expected "float"
```

```
nocheck_demo.py:8: error: Argument 2 to "Coordinate" has incompatible type "None"; expected "float"
```

As you can see, given the definition of `Coordinate`, Mypy knows that both arguments to create an instance must be of type `float`, but the assignment to `trash` uses a `str` and `None`.<sup>5</sup>

Now let's talk about the syntax and meaning of type hints.

## Variable annotation Syntax

Both `typing.NamedTuple` and `@dataclass` use the syntax of variable annotations defined in [PEP 526](#). This is quick introduction to that syntax in the context defining attributes in `class` statements.

The basic syntax of variable annotation is:

```
var_name: some_type
```

The type that goes after the `:` must be an identifier for one of these:

- a concrete class, for example `str` or `FrenchDeck`;
- an ABC—abstract base class;
- a type defined in the `typing` module, including special types and constructs like `Any`, `Optional`, `Union`, etc.;
- a type alias—as described in the [Type aliases](#) section of the [typing module documentation](#)

See [Acceptable type hints](#) in [PEP 484](#) for all details.

You can also initialize the variable with a value. In a `typing.NamedTuple` or `@dataclass` declaration, that value will



become the default for that attribute, if the corresponding argument is omitted in the constructor call.

```
var_name: some_type = a_value
```

## The meaning of variable annotations

We saw in “No runtime effect” that type hints have no effect at runtime. But at import time—when a module is loaded—Python does read them to build the `__annotations__` dictionary that `typing.NamedTuple` and `@dataclass` then use to enhance the class.

We’ll start this exploration with a simple class, so that we can later see what extra features are added by `typing.NamedTuple` and `@dataclass`.

*Example 5-10. meaning/demo\_plain.py: a plain class with type hints*

```
class DemoPlainClass:
```

```
    a: int          ❶  
    b: float = 1.1  ❷  
    c = 'spam'      ❸
```

- ❶ `a` becomes an annotation, but is otherwise discarded.
- ❷ `b` is saved as an annotation, and also becomes a class attribute with value `1.1`.
- ❸ `c` is just a plain old class attribute, not an annotation.

We can verify that in the console, first reading the `__annotations__` of the `DemoPlainClass`, then trying to get its attributes named `a`, `b`, and `c`:

```
>>> from demo_plain import DemoPlainClass  
>>> DemoPlainClass.__annotations__  
{'a': <class 'int'>, 'b': <class 'float'>}  
>>> DemoPlainClass.a
```

```

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'DemoPlainClass' has no attribute
'a'
>>> DemoPlainClass.b
1.1
>>> DemoPlainClass.c
'spam'

```

Note that the `__annotations__` special attribute is created by the interpreter to record the type hints that appear in the source code—even in a plain class.

The `a` survives only as an annotation. It doesn't become a class attribute because no value is bound to it.<sup>6</sup> The `b` and `c` are stored as class attributes because they are bound to values.

None of those three attributes will be in a new instance of `DemoPlainClass`. If you create an object `o = DemoPlainClass()`, `o.a` will raise `AttributeError`, while `o.b` and `o.c` will retrieve the class attributes with values `1.1` and `'spam'`—that's just normal Python object behavior.

## INSPECTING A `TYPING.NAMEDTUPLE`

Now let's examine a class built with `typing.NamedTuple`, using the same attributes and annotations as `DemoPlainClass` from [Example 5-10](#).

*Example 5-11. `meaning/demo_nt.py`: a class built with `typing.NamedTuple`.*

---

```

import typing

class DemoNTClass(typing.NamedTuple):
    a: int

```

```
b: float = 1.1 ②  
c = 'spam' ③
```

- ❶ a becomes an annotation and also an instance attribute.
- ❷ b is another annotation, and also becomes an instance attribute with default value 1.1.
- ❸ c is just a plain old class attribute; no annotation will refer to it.

Inspecting the DemoNTClass, we get:

```
>>> from demo_nt import DemoNTClass  
>>> DemoNTClass.__annotations__  
{'a': <class 'int'>, 'b': <class 'float'>}  
>>> DemoNTClass.a  
<_collections._tuplegetter object at 0x101f0f940>  
>>> DemoNTClass.b  
<_collections._tuplegetter object at 0x101f0f8b0>  
>>> DemoNTClass.c  
'spam'
```

Here we see the same annotations for a and b as we saw in [Example 5-10](#). But DemoNTClass has three class attributes a, b, and c. The c attribute is just a plain class attribute with the value 'spam'.

The a and b class attributes are actually *descriptors*—an advanced feature covered in [\[Link to Come\]](#). For now, think of them as similar to property getters: methods that don't require the explicit call operator ( ) to retrieve an instance attribute. In practice, this means a and b will work as read-only instance attributes—which makes sense when we recall that DemoNTClass instances are just a fancy tuples, and tuples are immutable.

DemoNTClass also gets a custom docstring:

```
>>> DemoNTClass.__doc__  
'DemoNTClass(a, b)'
```

Let's inspect an instance of `DemoNTClass`:

```
>>> nt = DemoNTClass(8)
>>> nt.a
8
>>> nt.b
1.1
>>> nt.c
'spam'
```

To construct `nt`, we need to give at least the `a` argument to `DemoNTClass`. The constructor also takes a `b` argument, but it has a default value of `1.1`, so it's optional. The `nt` object has the `a` and `b` attributes as expected; it doesn't have a `c` attribute, but Python retrieves it from the class, as usual.

If you try to assign values to `nt.a`, `nt.b`, `nt.c` or even `nt.z` you'll get `AttributeError`, with subtly different error messages. Try that and reflect on the messages.

## INSPECTING A CLASS DECORATED WITH `DATACLASS`

Now we'll examine Example 5-12:

*Example 5-12. meaning/demo\_dc.py: a class decorated with `@dataclass`*

---

```
from dataclasses import dataclass
```

```
@dataclass
```

```
class DemoDataClass:
```

```
    a: int           ❶
    b: float = 1.1   ❷
    c = 'spam'       ❸
```

❶ `a` becomes an annotation and also an instance attribute.

- ❷ `b` is another annotation, and also becomes an instance attribute with default value `1.1`.
- ❸ `c` is just a plain old class attribute; no annotation will refer to it.

Now let's check out `__annotations__`, `__doc__`, and the `a`, `b`, `c` attributes on `DemoDataClass`:

```
>>> from demo_dc import DemoDataClass
>>> DemoDataClass.__annotations__
{'a': <class 'int'>, 'b': <class 'float'>}
>>> DemoDataClass.__doc__
'DemoDataClass(a: int, b: float = 1.1)'
>>> DemoDataClass.a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'DemoDataClass' has no attribute 'a'
>>> DemoDataClass.b
1.1
>>> DemoDataClass.c
'spam'
```

The `__annotations__` and `__doc__` are not surprising. However, there is no attribute named `a` in `DemoDataClass`—in contrast with `DemoNTClass` from [Example 5-11](#), which has a descriptor to get `a` from the instances as read-only attributes (that mysterious `<_collections._tuplegetter>`). That's because the `a` attribute will only exist in instances of `DemoDataClass`. It will be a public attribute that we can get and set, unless the class is frozen. But `b` and `c` exist as class attributes, with `b` holding the default value for the `b` instance attribute, while `c` is just a class attribute that will not be bound to the instances.

Now let's see how a `DemoDataClass` instance looks like:

```
>>> dc = DemoDataClass(9)
>>> dc.a
```

```
9
>>> dc.b
1.1
>>> dc.c
'spam'
```

Again, `a` and `b` are instance attributes, and `c` is a class attribute we get via the instance.

As mentioned, `DemoDataClass` instances are mutable—and no type checking is done at runtime:

```
>>> dc.a = 10
>>> dc.b = 'oops'
```

We can do even sillier assignments:

```
>>> dc.c = 'whatever'
>>> dc.z = 'secret stash'
```

Now the `dc` instance has a `c` attribute—but that does not change the `c` class attribute. And we can add a new `z` attribute. This is normal Python behavior: regular instances can have their own attributes that don't appear in the class.<sup>7</sup>

## More about `@dataclass`

We've only seen simple examples of `@dataclass` use so far. The decorator accepts several arguments. This is its signature:

```
@dataclass(*, init=True, repr=True, eq=True, order=False,
            unsafe_hash=False, frozen=False)
```

The \* in the first position means the remaining parameters are keyword-only. Table 5-2 describes them.

*Table 5-2. Keyword parameters accepted by the @dataclass decorator*

option	default	meaning	notes
init	True	generate <code>__init__</code>	Ignored if <code>__init__</code> is implemented by user.
repr	True	generate <code>__repr__</code>	Ignored if <code>__repr__</code> is implemented by user.
eq	True	generate <code>__eq__</code>	Ignored if <code>__eq__</code> is implemented by user.
order	False	generate <code>__lt__</code> , <code>__le__</code> , <code>__gt__</code> , <code>__ge__</code>	Raises exceptions if <code>eq=False</code> or any of the listed special methods are implemented by user.
unsafe_hash	False	generate <code>__hash__</code>	Complex semantics and several caveats—see: <a href="#">dataclass documentation</a> .
frozen	False	make instances “immutable”	instances will be reasonably safe from accidental change, but not really immutable <sup>a</sup> .

<sup>a</sup> `@dataclass` emulates imutability by generating `__setattr__` and `__delattr__` which raise `dataclass.FrozenInstanceError`—a subclass of `AttributeError`—when the user attempts to set or delete a field.

The defaults are really the most useful settings for common use cases. The options you are more likely to change from the defaults are:

- `order=True`: to allow sorting of instances of the data class;

- `frozen=True`: to protect against accidental changes to the class instances.

Given the dynamic nature of Python objects, it's not too hard for a nosy programmer to go around the protection afforded by `frozen=True`. But the necessary tricks should be easy to spot on a code review.

If `eq` and `frozen` are both true, `@dataclass` will produce a suitable `__hash__` method, so the instances will be hashable. The generated `__hash__` will use data from all fields that are not individually excluded using a field option we'll see in “[Key-sharing dictionary](#)”. If `frozen=False` (the default), `@dataclass` will set `__hash__` to `None`, signalling that the instances are unhashable, therefore overriding `__hash__` from any superclass.

Regarding `unsafe_hash`, PEP 557 has this to say:

*Although not recommended, you can force Data Classes to create a `__hash__` method with `unsafe_hash=True`. This might be the case if your class is logically immutable but can nonetheless be mutated. This is a specialized use case and should be considered carefully.*

I will leave `unsafe_hash` at that. If you feel you must use that option, check the [dataclasses.dataclass](#) documentation.

Further customization of the generated data class can be done at a field level.

## Field options

We've already seen most basic field option: providing or not a default value with the type hint. Note that fields are read in order, and after you declare a field with a default value, all remaining fields must also have default values. This limitation makes sense: the fields will



become parameters in the generated `__init__`, and Python does not allow non-default parameters following parameters with defaults.

Mutable default values are a common source of bugs for beginning Python developers. In function definitions, a mutable default value is easily corrupted when one invocation of the function mutates the default, changing the behavior of further invocations—an issue we’ll explore in “[Mutable Types as Parameter Defaults: Bad Idea](#)” (Chapter 6). Class attributes are often used as default attribute values for instances, including in data classes. And `@dataclass` uses the default values in the type hints to generate parameters with defaults for `__init__`. To prevent bugs, `@dataclass` rejects the class definition in [Example 5-13](#).

*Example 5-13. `dataclass/club_wrong.py`: this class raises `ValueError`*

---

```
@dataclass
class ClubMember:

    name: str
    guests: list = []
```

If you load the module with that `ClubMember` class, this is what you get:

```
$ python3 club_wrong.py
Traceback (most recent call last):
  File "club_wrong.py", line 4, in <module>
    class ClubMember:
    ...several lines omitted...
ValueError: mutable default <class 'list'> for field guests
is not allowed:
use default_factory
```

The `ValueError` message explains the problem and suggests a solution: use `default_factory`. This is how to correct `ClubMember`:

*Example 5-14. `dataclass/club.py`: this `ClubMember` definition works.*

---

```
from dataclasses import dataclass, field

@dataclass
class ClubMember:

    name: str
    guests: list = field(default_factory=list)
```

In the `guests` field of [Example 5-14](#), instead of a literal list, the default value is set by calling the `dataclasses.field` function with `default_factory=list`. The `default_factory` parameter lets you provide a function, class, or any other callable, which will be invoked with zero arguments to build a default value each time an instance of the data class is created. This way, each instance of `ClubMember` will have its own `list`—instead of all instances sharing the same `list` from the class, which is rarely what we want and is often a bug.

### WARNING

It's good that `@dataclass` rejects class definitions with a `list` default value in a field. However, be aware that it is a partial solution that only applies to `list`, `dict` and `set`. Other mutable values used as defaults will not be flagged by `@dataclass`. It's up to you to understand the problem and remember to use a default factory to set mutable default values.

If you browse the `dataclasses` module documentation, you'll see a `list` field defined with a novel syntax, as in [Example 5-15](#).

*Example 5-15. `dataclass/club_generic.py`: this `ClubMember` definition is more precise*

---

```
from dataclasses import dataclass, field
from typing import List ❶

@dataclass
class ClubMember:

    name: str
    guests: List[str] = field(default_factory=list) ❷
```

- ❶ Import the `List` type from `typing`.
- ❷ `List[str]` means “a list of `str`”.

The new syntax `List[str]` is a generic type definition: the `List` class from `typing` accepts that bracket notation to specify the type of the list items. We’ll cover generics in [\[Link to Come\]](#). For now, note that both [Example 5-14](#) and [Example 5-15](#) are correct, and the `Mypy` type checker does not complain about either of those class definitions. But the second one is more precise, and will allow the type checker to verify code that puts items in the list, or that read items from it.

The `default_factory` is by far the most frequently used option of the `field` function, but there are several others, listed in [Table 5-3](#).

Table 5-3. Keyword arguments accepted by the `field` function

option	default	meaning
<code>default</code>	<code>_MISSING_TYPE</code>	default value for field <sup>a</sup>
<code>default_factory</code>	<code>_MISSING_TYPE</code>	0-parameter function used to produce a default
<code>init</code>	<code>True</code>	include field in parameters to <code>__init__</code>
<code>repr</code>	<code>True</code>	include field <code>__repr__</code>
<code>hash</code>	<code>None</code>	use field to compute <code>__hash__</code> ` footnote: [ <code>`hash=None</code> means the field will be used in <code>__hash__</code> only if <code>compare=True</code> .]
<code>compare</code>	<code>True</code>	use field in comparison methods <code>__eq__</code> , <code>__gt__</code> etc.
<code>metadata</code>	<code>None</code>	mapping with user-defined data; ignored by the <code>@dataclass</code>

<sup>a</sup> `dataclass._MISSING_TYPE` is a sentinel value indicating the option was not provided. It exists so we can set `None` as an actual default value, a common use case.

The `default` option exists because the `field` call takes the place of the default value in the field annotation. If you want to create an `athlete` field with default value of `False`, and also omit that field from the `__repr__` method, you'd write this:

```
@dataclass
class ClubMember:

    name: str
    guests: list = field(default_factory=list)
    athlete: bool = field(default=False, repr=False)
```

## Post-init processing

The `__init__` method generated by `@dataclass` only takes the arguments passed and assigns them—or their default values, if missing—to the instance attributes that are instance fields. But you may need to do more than that to initialize the instance. If that’s the case, you can provide a `__post_init__` method. When that method exists, `@dataclass` will add code to the generated `__init__` to call `__post_init__` as the last step.

Common use cases for `__post_init__` are validation and computing field values based on other fields. We’ll study a simple example that uses `__post_init__` for both of these reasons.

First, let’s look at the expected behavior of a `ClubMember` subclass named `HackerClubMember`, as described by doctests in [Example 5-16](#).

*Example 5-16. `dataclass/hackerclub.py`: doctests for `HackerClubMember`*

---

```
"""
`HackerClubMember` objects accept an optional `handle`
argument::
```

```
>>> anna = HackerClubMember('Anna Ravenscroft',
handle='AnnaRaven')
>>> anna
HackerClubMember(name='Anna Ravenscroft', guests=[],
handle='AnnaRaven')
```

```
If `handle` is omitted, it's set to the first part of the
member's name::
```

```
>>> leo = HackerClubMember('Leo Rochael')
>>> leo
HackerClubMember(name='Leo Rochael', guests=[],
handle='Leo')
```

*Members must have a unique handle. The following ``leo2`` will not be created, because its ``handle`` would be 'Leo', which was taken by ``leo``:*

```
>>> leo2 = HackerClubMember('Leo DaVinci')
Traceback (most recent call last):
...
ValueError: handle 'Leo' already exists.
```

*To fix, ``leo2`` must be created with an explicit ``handle``:*

```
>>> leo2 = HackerClubMember('Leo DaVinci', handle='Neo')
>>> leo2
HackerClubMember(name='Leo DaVinci', guests=[],
handle='Neo')
"""
```

Note that we must provide `handle` as a keyword argument, because `HackerClubMember` inherits `name` and `guests` from `ClubMember`, and adds the `handle` field. The generated docstring for `HackerClubMember` shows the order of the fields in the constructor call:

```
>>> HackerClubMember.__doc__
"HackerClubMember(name: str, guests: list = <factory>,
handle: str = '')"
```

Here, `<factory>` is a short way of saying that some callable will produce the default value for `guests` (in our case, the factory is the `list` class). The point is: to provide a `handle` but no `guests`, we must pass `handle` as a keyword argument.

The Inheritance section of the `dataclasses` module documentation explains how the order of the fields is computed when there are several levels of inheritance.

## NOTE

In [Link to Come] we'll talk about misusing inheritance, particularly when the superclasses are not abstract. Creating a hierarchy of data classes is usually a bad idea, but it served us well here to make [Example 5-17](#) shorter, focusing on the `handle` field declaration and `__post_init__` validation.

[Example 5-17](#) is the implementation:

*Example 5-17. `dataclass/hackerclub.py`: code for `HackerClubMember`.*

```
from dataclasses import dataclass
from club import ClubMember

@dataclass
class HackerClubMember(ClubMember): ❶

    all_handles = set() ❷

    handle: str = '' ❸

    def __post_init__(self):
        cls = self.__class__ ❹
        if self.handle == '': ❺
            self.handle = self.name.split()[0]
        if self.handle in cls.all_handles: ❻
            msg = f'handle {self.handle!r} already exists.'
            raise ValueError(msg)
        cls.all_handles.add(self.handle) ❼
```

- ❶ `HackerClubMember` extends `ClubMember`.
- ❷ `all_handles` is a class attribute.
- ❸ `handle` is an instance field of type `str` with empty string as its default value; this makes it optional.

- ④ Get the class of the instance.
- ⑤ If `self.handle` is the empty string, set it to the first part of `name`.
- ⑥ If `self.handle` is in `cls.all_handles`, raise `ValueError`.
- ⑦ Add the new handle to `cls.all_handles`.

[Example 5-17](#) works as intended, but is not satisfactory to a static type checker. Next, we'll see why, and how to fix it.

## Typed class attributes

If we typecheck [Example 5-17](#) with Mypy, we are reprimanded:

```
$ mypy hackerclub.py
hackerclub.py:38: error: Need type annotation for
'all_handles'
(hint: "all_handles: Set[<type>] = ...")
Found 1 error in 1 file (checked 1 source file)
```

Unfortunately, the hint provided by Mypy (version 0.750 as I write this) is not helpful in the context of `@dataclass` usage. If we add a type hint like `Set[...]` to `all_handles`, `@dataclass` will find that annotation and make `all_handles` an instance field. We saw this happening in [“Inspecting a class decorated with `dataclass`”](#).

The work-around defined in [PEP 526—Syntax for Variable Annotations](#) is a *class variable annotation*, written with a pseudo-type named `typing.ClassVar`, which leverages the generics `[]` notation to set the type of the variable and also declare it a class attribute.

To make the type checker happy, this is how we are supposed to declare `all_handles` in [Example 5-17](#):

```
all_handles: ClassVar[Set[str]] = set()
```



That type hint is saying:

*all\_handles is a class attribute of type set-of-str, with an empty set as its default value.*

To code that annotation, we must import `ClassVar` and `Set` from the `typing` module.

The `@dataclass` decorator doesn't care about the types in the annotations, except in two cases, and this is one of them: if the type is `ClassVar`, an instance field will not be generated for that attribute.

The other case where the type of the field is relevant to `@dataclass` is when declaring *init-only variables*, our next topic.

## Initialization variables that are not fields

Sometimes you may need to pass arguments to `__init__` that are not instance fields. Such arguments are called *init-only variables* by the [dataclasses documentation](#). To declare an argument like that, `dataclasses` module provides the pseudo-type `InitVar`, which uses the same syntax of `typing.ClassVar`. The example given in the documentation is a data class that has a field initialized from a database, and the database object must be passed to the constructor.

This is the code that illustrates the [Init-only variables](#) section:

*Example 5-18. Example from the [dataclasses](#) module documentation.*

---

```
@dataclass
class C:
    i: int
    j: int = None
    database: InitVar[DatabaseType] = None

    def __post_init__(self, database):
```

```
if self.j is None and database is not None:
    self.j = database.lookup('j')
```

```
c = C(10, database=my_database)
```

Note how the `database` attribute is declared. `InitVar` will prevent `@dataclass` from treating `database` as a regular field. It will not be set as an instance attribute, and the `dataclasses.fields` function will not list it. However, `database` will be one of the arguments that the generated `__init__` will accept, and it will be also passed to `__post_init__`—if you write that method, you must add a corresponding argument to the method signature, as shown in [Example 5-18](#)

This rather long overview of `@dataclass` covered the most useful features—some of them appeared in previous sections, like “[Main features](#)” where we covered all three data class builders in parallel. The [dataclasses](#) documentation and [PEP 526 — Syntax for Variable Annotations](#) have all details.

## @dataclass Example: Dublin Core Resource Record

Often, classes built with `@dataclass` will have more fields than the very short examples presented so far. Dublin Core provides the foundation for a more typical `@dataclass` example.

*The Dublin Core Schema is a small set of vocabulary terms that can be used to describe digital resources (video, images, web pages, etc.), as well as physical resources such as books or CDs, and objects like artworks.*

—Dublin Core on Wikipedia

The standard defines 15 optional fields, the `Resource` class in [Example 5-19](#) uses 8 of them.

*Example 5-19. dataclass/resource.py: code for Resource, a class based on Dublin Core terms.*

---

```
from dataclasses import dataclass, field
from typing import List, Optional
from enum import Enum, auto
from datetime import date

class ResourceType(Enum): ❶
    BOOK = auto()
    EBOOK = auto()
    VIDEO = auto()

@dataclass
class Resource:
    """Media resource description."""
    identifier: str ❷
    title: str = '<untitled>' ❸
    creators: List[str] = field(default_factory=list)
    date: Optional[date] = None ❹
    type: ResourceType = ResourceType.BOOK ❺
    description: str = ''
    language: str = ''
    subjects: List[str] = field(default_factory=list)
```

- ❶ This Enum will provide type-safe values for the `Resource.type` field.
- ❷ `identifier` is the only required field.
- ❸ `title` is the first field with a default. This forces all fields below to provide defaults.
- ❹ The value of `date` can be a `datetime.date` instance, or `None`.
- ❺ The type field default is `ResourceType.BOOK`.

Example 5-20 is a doctest to demonstrate how a `Resource` record appears in code:

*Example 5-20. dataclass/resource.py: code for Resource, a class based on Dublin Core terms.*

---

```
>>> description = 'Improving the design of existing code'
>>> book = Resource('978-0-13-475759-9', 'Refactoring, 2nd
Edition',
...     ['Martin Fowler', 'Kent Beck'], date(2018, 11, 19),
...     ResourceType.BOOK, description, 'EN',
...     ['computer programming', 'OOP'])
>>> book # doctest: +NORMALIZE_WHITESPACE
Resource(identifier='978-0-13-475759-9', title='Refactoring,
2nd Edition',
creators=['Martin Fowler', 'Kent Beck'],
date=datetime.date(2018, 11, 19),
type=<ResourceType.BOOK: 1>, description='Improving the
design of existing code',
language='EN', subjects=['computer programming', 'OOP'])
```

The `__repr__` generated by `@dataclass` is OK, but we can do better. This is the format we want from `repr(book)`:

```
>>> book # doctest: +NORMALIZE_WHITESPACE
Resource(
    identifier = '978-0-13-475759-9',
    title = 'Refactoring, 2nd Edition',
    creators = ['Martin Fowler', 'Kent Beck'],
    date = datetime.date(2018, 11, 19),
    type = <ResourceType.BOOK: 1>,
    description = 'Improving the design of existing
code',
    language = 'EN',
    subjects = ['computer programming', 'OOP'],
)
```

Example 5-21 is the code of `__repr__` to produce the format above. This example uses `doctest.fields` to get the names of the data class fields.

*Example 5-21. dataclass/resource\_repr.py: code for `__repr__` method implemented in the `Resource` class from Example 5-19.*

---

```
def __repr__(self):
    cls = self.__class__
    cls_name = cls.__name__
    indent = ' ' * 4
    res = [f'{cls_name}(']
    for f in fields(cls):
        value = getattr(self, f.name)
        res.append(f'{indent}{f.name} = {value!r},')

    res.append(')')
    return '\n'.join(res)
```

- ❶ Start the `res` list to build the output string with the class name and open parenthesis.
- ❷ For each field `f` in the class...
- ❸ Get the named attribute from the instance.
- ❹ Append an indented line with the name of the field and `repr(value)`—that's what the `!r` does.
- ❺ Append closing parenthesis.
- ❻ Build multiline string from `res` and return it.

With this example inspired by the soul of Dublin, Ohio, we conclude our tour of Python's data class builders.

Data classes are handy, but your project may suffer if you overuse them. The next section explains.

## Data class as a code smell

Whether you implement a data class writing all the code yourself or leveraging one of the class builders described in this chapter, be aware that it may signal a problem in your design.

In *Refactoring, Second Edition*, Martin Fowler and Kent Beck present a catalog of “code smells”—patterns in code that may indicate the need for refactoring. The entry titled *Data Class* starts like this:

*These are classes that have fields, getting and setting methods for fields, and nothing else. Such classes are dumb data holders and are often being manipulated in far too much detail by other classes.*

In Fowler’s personal Web site there’s an illuminating post explaining what is a “code smell”. That post is very relevant to our discussion because he uses *data class* as one example of a code smell and suggests how to deal with it. Here is the post, reproduced in full.<sup>8</sup>

## CODE SMELL

**By Martin Fowler**

A code smell is a surface indication that usually corresponds to a deeper problem in the system. The term was first coined by Kent Beck while helping me with my Refactoring book.

The quick definition above contains a couple of subtle points. Firstly a smell is by definition something that's quick to spot—or sniffable as I've recently put it. A long method is a good example of this—just looking at the code and my nose twitches if I see more than a dozen lines of Java.

The second is that smells don't always indicate a problem. Some long methods are just fine. You have to look deeper to see if there is an underlying problem there—smells aren't inherently bad on their own—they are often an indicator of a problem rather than the problem themselves.

The best smells are something that's easy to spot and most of time lead you to really interesting problems. Data classes (classes with all data and no behavior) are good examples of this. You look at them and ask yourself what behavior should be in this class. Then you start refactoring to move that behavior in there. Often simple questions and initial refactorings can be the vital step in turning anemic objects into something that really has class.

One of the nice things about smells is that it's easy for inexperienced people to spot them, even if they don't know enough to evaluate if there's a real problem or to correct them. I've heard of lead developers who will pick a "smell of the week" and ask people to look for the smell and bring it up with the senior members of the team. Doing it one smell at a time is a good way of gradually teaching people on the team to be better programmers.

The main idea of Object Oriented Programming is to place behavior and data together in the same code unit: a class. If a class is widely used but has no significant behavior of its own, it's possible that code dealing with its instances is scattered (and even duplicated) in methods and functions throughout the system—a recipe for maintenance headaches. That's why Fowler's refactorings to deal with a data class involve bringing responsibilities back into it.

Taking that into account, there are a couple of common scenarios where it makes sense to have a data class with little or no behavior.

## **Data class as scaffolding**

In this scenario, the data class is an initial, simplistic implementation of a class to jump start a new project or module. With time, the class should get its own methods, instead of relying on methods of other classes to operate on its instances. Scaffolding is temporary; eventually your custom class may become fully independent from the builder you used to start it.

Python is also used for quick problem solving and experimentation, and then it's OK leave the scaffolding in place.

## **Data class as intermediate representation**

A data class can be useful to build records about to be exported to JSON or some other interchange format, or to hold data that was just imported, crossing some system boundary. Python's data class builders all provide a method or function to convert an instance to a plain `dict`, and you can always invoke the constructor with a `dict` used as keyword arguments expanded with `**`. Such a `dict` is very close to a JSON record.

In this scenario, the data class instances should be handled as immutable objects—even if the fields are mutable, you should not change them while they are in this intermediate form. If you do, you're losing the key benefit of having data and behavior close together. When importing/exporting requires changing values, you should implement your own builder methods instead of using the given “as dict” methods or standard constructors.

After reviewing Python's data class builders, we'll end the chapter with the `struct` module, also used for importing/exporting records,



but at a much lower level.

## Parsing binary records with struct

The `struct` module provides functions to parse fields of bytes into a tuple of Python objects, and to perform the opposite conversion, from a tuple into packed bytes. `struct` can be used with `bytes`, `bytearray`, and `memoryview` objects.

Suppose you need to read a binary file containing data about metropolitan areas, produced by a program in C with a record defined as [Example 5-22](#)

*Example 5-22. MetroArea: a struct in the C language.*

---

```
struct MetroArea {  
    int year;  
    char name[12];  
    char country[2];  
    float population;  
};
```

Here is how to read one record in that format, using `struct.unpack`:

*Example 5-23. Reading a C struct in the Python console.*

---

```
>>> from struct import unpack  
>>> FORMAT = 'i12s2sf'  
>>> data = open('metro_areas.bin', 'rb').read(24)  
>>> data  
b"\xe2\x07\x00\x00Tokyo\x00\xc5\x05\x01\x00\x00\x00JP\x00\x00\x1  
1X'L"  
>>> unpack(FORMAT, data)  
(2018, b'Tokyo\x00\xc5\x05\x01\x00\x00\x00', b'JP', 43868228.0)
```

Note how `unpack` returns a tuple with four fields, as specified by the `FORMAT` string. The letters and numbers in `FORMAT` are [Format Characters](#) described in the `struct` module documentation.

Table 5-4 explains the elements of the format string from Example 5-23.

Table 5-4. Parts of the format string 'i12s2sf'.

part	size	C type	Python type	limits to actual content
i	4 bytes	int	int	32 bits; range -2,147,483,648 to 2,147,483,647
12s	12 bytes	char[12]	bytes	length = 12
2s	2 bytes	char[2]	bytes	length = 2
f	4 bytes	float	float	32-bits; approximate range $\pm 3.4 \times 10^{38}$

One detail about the layout of `metro_areas.bin` is not clear from the code in Example 5-22: size is not the only difference between the `name` and `country` fields. The `country` field always holds a 2-letter country code, but `name` is a null-terminated sequence with up to 12 bytes including the terminating `b'\0'`—which you can see in Example 5-23 right after the word Tokyo.<sup>9</sup>

Now let's review a script to extract all records from `metro_areas.bin` and produce a simple report like this:

```
$ python3 metro_read.py
2018 Tokyo, JP 43,868,228
2015 Shanghai, CN 38,700,000
2015 Jakarta, ID 31,689,592
```

Example 5-24 showcases the handy `struct.iter_unpack` function.

Example 5-24. *metro\_read.py*: list all records from *metro\_areas.bin*

---

```
from struct import iter_unpack

FORMAT = 'i12s2sf'

def text(field: bytes) -> str:
    octets = field.split(b'\0', 1)[0]
    return octets.decode('cp437')

with open('metro_areas.bin', 'rb') as fp:
    data = fp.read()

for fields in iter_unpack(FORMAT, data):
    year, name, country, pop = fields
    place = text(name) + ', ' + text(country)
    print(f'{year}\t{place}\t{pop:,.0f}')
```

- ❶ The struct format.
- ❷ Utility function to decode and clean up the bytes fields; returns a `str`.<sup>10</sup>
- ❸ Handle null-terminated C string: split once on `b'\0'`, then take the first part.
- ❹ Decode bytes into `str`.
- ❺ Open and read the whole file in binary mode; `data` is a bytes object.
- ❻ `iter_unpack(...)` returns a generator that produces one tuple of fields for each sequence of bytes matching the format string.
- ❼ The name and country fields need further processing by the `text` function.

The `struct` module provides no way to specify null-terminated string fields. When processing a field like `name` in the example above, after unpacking we need to inspect the returned bytes to discard the first `b'\0'` and all bytes after it in that field. It is quite possible that bytes after the first `b'\0'` and up to the end of the field are garbage. You can actually see that in [Example 5-23](#).

Memory views can make it easier to experiment and debug programs using `struct`, as the next section explains.

## Structs and Memory Views

We saw in “Memory Views” that the `memoryview` type does not let you create or store byte sequences, but provides shared memory access to slices of data from other binary sequences, packed arrays, and buffers such as Python Imaging Library (PIL) images,<sup>11</sup> without copying the bytes.

Example 5-25 shows the use of `memoryview` and `struct` together to extract the width and height of a GIF image.

*Example 5-25. Using `memoryview` and `struct` to inspect a GIF image header*

---

```
>>> import struct
>>> fmt = '<3s3sHH' ❶
>>> with open('filter.gif', 'rb') as fp:
...     img = memoryview(fp.read()) ❷
...
>>> header = img[:10] ❸
>>> bytes(header) ❹
b'GIF89a+\x02\xe6\x00'
>>> struct.unpack(fmt, header) ❺
(b'GIF', b'89a', 555, 230)
>>> del header ❻
>>> del img
```

- ❶ `struct` format: `<` little-endian; `3s3s` two sequences of 3 bytes; `HH` two 16-bit integers.
- ❷ Create `memoryview` from file contents in memory...
- ❸ ...then another `memoryview` by slicing the first one; no bytes are copied here.
- ❹ Convert to `bytes` for display only; 10 bytes are copied here.
- ❺

Unpack `memoryview` into tuple of: type, version, width, and height.

- 6 Delete references to release the memory associated with the `memoryview` instances.

Note that slicing a `memoryview` returns a new `memoryview`, without copying bytes.<sup>12</sup>

I will not go deeper into `memoryview` or the `struct` module, but if you work with binary data, you'll find it worthwhile to study their docs: [Built-in Types » Memory Views](#) and [struct — Interpret bytes as packed binary data](#).

## Should we use `struct`?

I only recommend `struct` to handle data from legacy systems or standardized binary formats.

Proprietary binary records in the real world are brittle and can be corrupted easily. Our super simple example exposed one of many caveats: a string field may be limited only by its size in bytes, it may be padded by spaces, or it may contain a null-terminated string followed by random garbage up to a certain size. There is also the problem of endianness: the order of the bytes used to represent integers and floats, which depends on the CPU architecture.

If you need to exchange binary data among Python systems, the `pickle` module is the easiest way, by far. If the exchange involves programs in other languages, use JSON or a multi-platform binary serialization format like [MessagePack](#) or [Protocol Buffers](#).

## Chapter Summary

The main topic of this chapter were the data class builders `collections.namedtuple`, `typing.NamedTuple` and `dataclasses.dataclass`. We saw that each of them generate data classes from descriptions provided as arguments to a factory function or from `class` statements with type hints—in the case of the latter two. In particular, both named tuple variants produce `tuple` subclasses, adding only the ability to access fields by name, and providing a `_fields` class attribute listing the field names as a tuple of strings.

Next we studied the main features of the three class builders side by side, including how to extract instance data as a `dict`, how to get the names and default values of fields, and how to make a new instance from an existing one.

This prompted our first look into type hints, particularly those used to annotate attributes in a `class` statement, using the notation introduced in Python 3.6 with PEP 526—Syntax for Variable Annotations. Probably the most surprising aspect of type hints in general is the fact that they have no effect at all at runtime. Python remains a dynamic language. External tools, like Mypy, are needed to take advantage of typing information do detect errors via static analysis of the source code. After a basic overview of the syntax from PEP 526, we studied the effect of annotations in a plain class and in classes built by `typing.NamedTuple` and `@dataclass`.

Next we covered the most commonly used features provided by `@dataclass` and the `default_factory` option of the `dataclasses.field` function. We also looked into the special pseudo-type hints `typing.ClassVar` and `dataclasses.InitVar` that are important in the context of data classes. This main topic concluded with an example based on the Dublin Core Schema, which illustrated how to use `dataclasses.fields` to iterate over the attributes of a `Resource` instance in a custom `__repr__`.

The “[Data class as a code smell](#)” came after that, warning against possible abuse of data classes defeating a basic principle of Object Oriented Programming: data and the functions that touch it should be together in the same class. Classes with no logic may be a sign of misplaced logic.

The final topic was a brief overview of the `struct` package, used to read and write records encoded as packed byte sequences. That section ended with suggestion for alternative libraries for binary data exchange, including Python’s `pickle` and language-independend alternatives.

## Further Reading

Python’s standard documentation for the data class builders we covered is very good, and has quite a few small examples.

For `@dataclass` in particular, most of [PEP 557—Data Classes](#) was copied into the `dataclasses` module documentaion. But [PEP 557](#) has a few very informative sections that were not copied, including [Why not just use namedtuple?](#), [Why not just use typing.NamedTuple?](#) and the [Rationale](#) section which concludes with this Q&A:

*Where is it not appropriate to use Data Classes?*

*API compatibility with tuples or dicts is required. Type validation beyond that provided by PEPs 484 and 526 is required, or value validation or conversion is required.*

—Eric V. Smith, PEP 557 Rationale

Over at [RealPython.com](#), Geir Arne Hjelle wrote a very complete [Ultimate Guide to Data Classes in Python 3.7](#).

At PyCon US 2018, Raymond Hettinger presented [Dataclasses: The code generator to end all code generators](#) (video).

For more features and advanced functionality, including validation, the [attrs](#) project led by Hynek Schlawack goes way beyond `dataclasses`, promising “classes without boilerplate” and to “bring back the joy of writing classes by relieving you from the drudgery of implementing object protocols (aka dunder methods).” The influence of `attrs` on `@dataclass` is acknowledged by Eric V. Smith in PEP 557. This probably includes Smith’s most important architectural decision: the use of a class decorator instead of inheritance and/or a metaclass to do the job.

Glyph—founder of the Twisted project—wrote an excellent introduction to `attrs` in [The One Python Library Everyone Needs](#).

The `attrs` documentation includes a [discussion of alternatives](#).

Regarding *Data Class* as a code smell, the best source I found was Martin Fowler’s book *Refactoring, Second Edition*. This newest version is missing the quote from the epigraph of this chapter, “Data classes are like children...”, but otherwise it’s the best edition of Fowler’s most famous book, particularly for Pythonistas because the examples are in modern JavaScript, which is closer to Python than Java—the language of the first edition.

The Web site [Refactoring Guru](#) also has a description of the [Data Class](#) code smell.

For the `struct` module, again the [Python docs](#) are good and include simple examples. But before using `struct` you should seriously consider whether it’s feasible to use Python’s [json](#) or [pickle](#), or multi-language binary serialization libraries such as [MessagePack](#) and [Protocol Buffers](#).



## SOAPBOX

The entry for “[Guido](#)” in the Jargon file is about Guido van Rossum. It says, among other things:

*Mythically, Guido's most important attribute besides Python itself is Guido's time machine, a device he is reputed to possess because of the unnerving frequency with which user requests for new features have been met with the response “I just implemented that last night...”*

For the longest time, one of the missing pieces in Python's syntax has been a quick, standard way to declare instance attributes in a class. Many Object-Oriented languages have that. Here is part of a `Point` class definition in Smalltalk:

```
Object subclass: #Point
  instanceVariableNames: 'x y'
  classVariableNames: ''
  package: 'Kernel-BasicObjects'
```

The second line lists the names of the instance attributes `x` and `y`. If there were class attributes, they would be in the third line.

Python has always offered an easy way to declare class attributes, if they have an initial value. But instance attributes are much more common, and Python coders have been forced to look into the `__init__` method to find them, always afraid that there may be instance attributes created elsewhere in the class—or even created by external functions or methods of other classes.

Now we have `@dataclass`, yay!

But they bring their own problems.

First: when you use `@dataclass`, type hints are not optional. We've been promised for the last 7 years since [PEP 484—Type Hints](#) that they would always be optional. Now we have a major new language feature that requires them. If you don't like the whole static typing trend, you may want to use `attrs` instead.

Second: the [PEP 526](#) syntax for annotating instance and class attributes reverses the established convention of `class` statements: everything declared at the top-level of a `class` block was a class attribute (methods are class attributes too). With PEP 526 and `@dataclass`, any attribute declared at the top level with a type hint becomes an instance attribute. If it has a default value, then it will also be a class attribute. But if you write `limit = 99` at the top-level

of a dataclass, with no type hints, suddenly you are back in the realm where declarations at the top-level of the class belong to the class. Finally, if you want to annotate that class attribute with a type, you can't use regular types because then it will become an instance attribute. You must resort to that pseudo-type `ClassVar` annotation:

```
limit: ClassVar[int] = 99
```

Here we are talking about the exception to the exception to the exception to the rule. This seems rather unpythonic to me.

I did not take part in the discussions leading to PEP 526 or [PEP 557—Data Classes](#), but here is an alternative syntax that I'd like to see:

```
@dataclass
class HackerClubMember:

    .name: str
    .guests: list = field(default_factory=list)
    .handle: str = ''

    all_handles = set()
```

- ❶ Instance attributes must be declared with a `.` prefix. The dot is not part of the name—it's only used in this context to denote an instance attribute.
- ❷ Any attribute name that doesn't have a `.` prefix is a class attribute (as they always have been).

The parser would have to change to accept that. I find this quite readable, and it avoids the triple-exception-to-the-rule issue.

I wish I could borrow Guido's time machine to go back to 2017 and sell this idea to the core team.

- 
- ❶ From *Refactoring, First Edition*, chapter 3, *Bad Smells in Code*, *Data Class* section, page 87.
  - ❷ Metaclasses are one of the subjects covered in [\[Link to Come\]](#)—*Class Metaprogramming*.

- 3 Class decorators are covered in [Link to Come]—*Class Metaprogramming*, along with metaclasses. Both are ways of customizing class behavior beyond what is possible with inheritance.
- 4 If you know Ruby, you know that injecting methods is a well-known but controversial technique among Rubyists. In Python, it's not as common, because it doesn't work with any built-in type—`str`, `list`, etc. I consider this limitation of Python a blessing.
- 5 In the context of type hints, `None` is not the `NoneType` singleton, but an alias for `NoneType` itself. This is strange when we stop to think about it, but appeals to our intuition and makes function return annotations easier to read in the common case of functions that return `None`.
- 6 Python has no concept of *undefined*, one of the silliest mistakes in the design of JavaScript. Thank Guido!
- 7 However, almost always when I see this in real code it's a bad idea. I once spent hours chasing a bug that was caused by attributes sneakily stashed in instances, like contraband across module borders. Also, setting an attribute after `__init__` may have a high memory cost, defeating the `__dict__` optimization we saw in "[Key-sharing dictionary](#)".
- 8 I am fortunate to have Martin Fowler as a colleague at ThoughtWorks, so it took just 20 minutes to get his permission.
- 9 `\0` and `\x00` are two valid escape sequences for the null character, `chr(0)`, in a Python `str` or `bytes` literal.
- 10 This is the first example using type hints in a function signature in this book. Simple type hints like these are quite readable and almost intuitive.
- 11 [Pillow](#) is PIL's most active fork.
- 12 Leonardo Rochaël—one of the technical reviewers—pointed out that even less byte copying would happen if I used the `mmap` module to open the image as a memory-mapped file. That module is outside the scope of this book, but if you read and change binary files frequently, learning more about [mmap — Memory-mapped file support](#) will be very fruitful.



# Chapter 6. Object References, Mutability, and Recycling

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at [fluentpython2e@ramalho.org](mailto:fluentpython2e@ramalho.org).

*‘You are sad,’ the Knight said in an anxious tone: ‘let me sing you a song to comfort you. [...] The name of the song is called “HADDOCKS’ EYES”.’*

*‘Oh, that’s the name of the song, is it?’ Alice said, trying to feel interested.*

*‘No, you don’t understand,’ the Knight said, looking a little vexed. ‘That’s what the name is CALLED. The name really IS “THE AGED AGED MAN.”’ (adapted from Chapter VIII. ‘It’s my own Invention’).*

—Lewis Carroll, *Through the Looking-Glass, and  
What Alice Found There*

Alice and the Knight set the tone of what we will see in this chapter. The theme is the distinction between objects and their names. A name is not the object; a name is a separate thing.

We start the chapter by presenting a metaphor for variables in Python: variables are labels, not boxes. If reference variables are old news to you, the analogy may still be handy if you need to explain aliasing issues to others.

We then discuss the concepts of object identity, value, and aliasing. A surprising trait of tuples is revealed: they are immutable but their values may change. This leads to a discussion of shallow and deep copies. References and function parameters are our next theme: the problem with mutable parameter defaults and the safe handling of mutable arguments passed by clients of our functions.

The last sections of the chapter cover garbage collection, the `del` command, and how to use weak references to “remember” objects without keeping them alive.

This is a rather dry chapter, but its topics lie at the heart of many subtle bugs in real Python programs.

## What’s new in this chapter

The topics covered here are very fundamental and stable. There were no changes since Python 3.4 that are worth mentioning in this 2<sup>nd</sup> edition.

I added a an example of using `is` to test for a sentinel object, and a warning about misuses of the `is` operator at the end of “Choosing Between `==` and `is`”.

This chapter used to be in Part IV, but I decided to bring it up earlier because it works better as an ending to Part II—*Data Structures*—than an opening to *Object-Oriented Idioms*.

Let’s start by unlearning that a variable is like a box where you store data.

## Variables Are Not Boxes

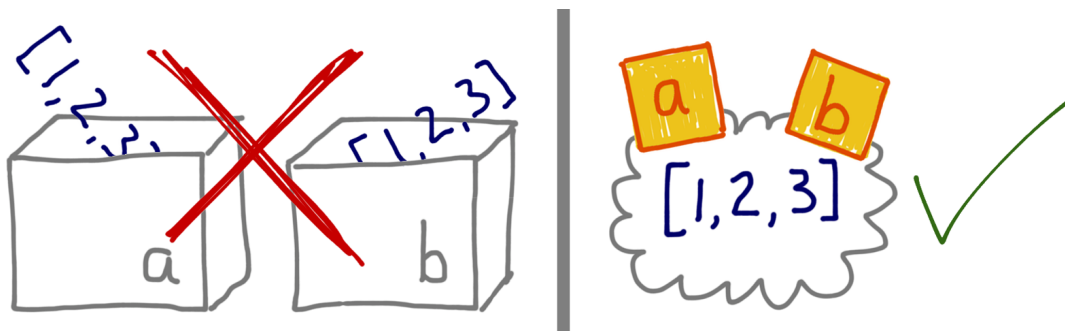
In 1997, I took a summer course on Java at MIT. The professor, Lynn Andrea Stein—an award-winning computer science educator who currently teaches at Olin College of Engineering—made the point that the usual “variables as boxes” metaphor actually hinders the understanding of reference variables in OO languages. Python variables are like reference variables in Java, so it’s better to think of them as labels attached to objects.

Example 6-1 is a simple interaction that the “variables as boxes” idea cannot explain. Figure 6-1 illustrates why the box metaphor is wrong for Python, while sticky notes provide a helpful picture of how variables actually work.

*Example 6-1. Variables *a* and *b* hold references to the same list, not copies of the list*

---

```
>>> a = [1, 2, 3]
>>> b = a
>>> a.append(4)
>>> b
[1, 2, 3, 4]
```



*Figure 6-1. If you imagine variables are like boxes, you can’t make sense of assignment in Python; instead, think of variables as sticky notes—Example 6-1 then becomes easy to explain*

Prof. Stein also spoke about assignment in a very deliberate way. For example, when talking about a seesaw object in a simulation, she would say: “Variable *s* is assigned to the seesaw,” but never “The

seesaw is assigned to variable s.” With reference variables, it makes much more sense to say that the variable is assigned to an object, and not the other way around. After all, the object is created before the assignment. [Example 6-2](#) proves that the righthand side of an assignment happens first.

*Example 6-2. Variables are assigned to objects only after the objects are created*

---

```
>>> class Gizmo:
...     def __init__(self):
...         print('Gizmo id: %d' % id(self))
...
>>> x = Gizmo()
Gizmo id: 4301489152 ❶
>>> y = Gizmo() * 10 ❷
Gizmo id: 4301489432 ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for *: 'Gizmo' and 'int'
>>>
>>> dir() ❹
['Gizmo', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'x']
```

- ❶ The output `Gizmo id: ...` is a side effect of creating a `Gizmo` instance.
- ❷ Multiplying a `Gizmo` instance will raise an exception.
- ❸ Here is proof that a second `Gizmo` was actually instantiated before the multiplication was attempted.
- ❹ But variable `y` was never created, because the exception happened while the right-hand side of the assignment was being evaluated.



## TIP

To understand an assignment in Python, always read the right-hand side first: that's where the object is created or retrieved. After that, the variable on the left is bound to the object, like a label stuck to it. Just forget about the boxes.

Because variables are mere labels, nothing prevents an object from having several labels assigned to it. When that happens, you have *aliasing*, our next topic.

## Identity, Equality, and Aliases

Lewis Carroll is the pen name of Prof. Charles Lutwidge Dodgson. Mr. Carroll is not only equal to Prof. Dodgson: they are one and the same. [Example 6-3](#) expresses this idea in Python.

*Example 6-3. charles and lewis refer to the same object*

---

```
>>> charles = {'name': 'Charles L. Dodgson', 'born': 1832}
>>> lewis = charles ❶
>>> lewis is charles
True
>>> id(charles), id(lewis) ❷
(4300473992, 4300473992)
>>> lewis['balance'] = 950 ❸
>>> charles
{'name': 'Charles L. Dodgson', 'balance': 950, 'born': 1832}
```

- ❶ lewis is an alias for charles.
- ❷ The is operator and the id function confirm it.
- ❸ Adding an item to lewis is the same as adding an item to charles.

However, suppose an impostor—let’s call him Dr. Alexander Pedachenko—claims he is Charles L. Dodgson, born in 1832. His credentials may be the same, but Dr. Pedachenko is not Prof. Dodgson. Figure 6-2 illustrates this scenario.

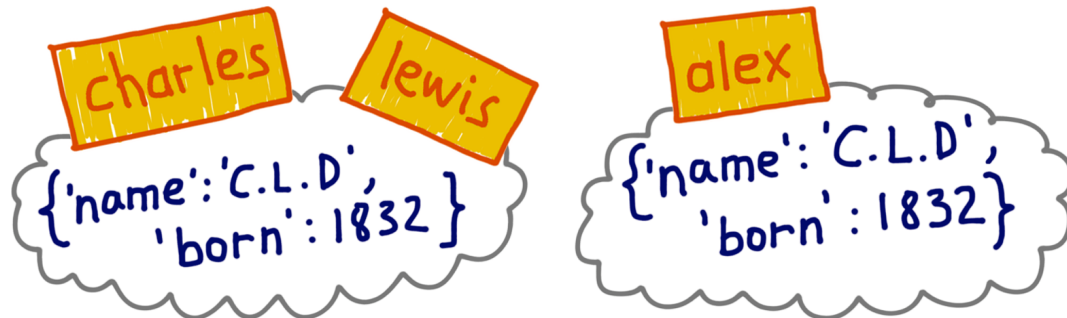


Figure 6-2. *charles* and *lewis* are bound to the same object; *alex* is bound to a separate object of equal contents

Example 6-4 implements and tests the *alex* object depicted in Figure 6-2.

*Example 6-4. alex and charles compare equal, but alex is not charles*

```
>>> alex = {'name': 'Charles L. Dodgson', 'born': 1832,
'balance': 950} ❶
>>> alex == charles ❷
True
>>> alex is not charles ❸
True
```

- ❶ *alex* refers to an object that is a replica of the object assigned to *charles*.
- ❷ The objects compare equal, because of the `__eq__` implementation in the `dict` class.
- ❸ But they are distinct objects. This is the Pythonic way of writing the negative identity comparison: `a is not b`.

Example 6-3 is an example of *aliasing*. In that code, *lewis* and *charles* are aliases: two variables bound to the same object. On the other hand, *alex* is not an alias for *charles*: these variables are bound to distinct objects. The objects bound to *alex* and *charles*

have the same *value*—that’s what `==` compares—but they have different identities.

In *The Python Language Reference*, “3.1. Objects, values and types” states:

*Every object has an identity, a type and a value. An object’s identity never changes once it has been created; you may think of it as the object’s address in memory. The `is` operator compares the identity of two objects; the `id()` function returns an integer representing its identity.*

The real meaning of an object’s ID is implementation-dependent. In CPython, `id()` returns the memory address of the object, but it may be something else in another Python interpreter. The key point is that the ID is guaranteed to be a unique numeric label, and it will never change during the life of the object.

In practice, we rarely use the `id()` function while programming. Identity checks are most often done with the `is` operator, and not by comparing IDs. Next, we’ll talk about `is` versus `==`.

## Choosing Between `==` and `is`

The `==` operator compares the values of objects (the data they hold), while `is` compares their identities.

We often care about values and not identities, so `==` appears more frequently than `is` in Python code.

However, if you are comparing a variable to a singleton, then it makes sense to use `is`. By far, the most common case is checking whether a variable is bound to `None`. This is the recommended way to do it:

```
x is None
```

And the proper way to write its negation is:

```
x is not None
```

`None` by far the most common singleton we test with `is`. Sentinel objects are another example of singletons we test with `is`. Here is how you could create and test a sentinel object:

```
END_OF_DATA = object()
# ... many lines
def traverse(...):
    # ... more lines
    if node is END_OF_DATA:
        raise StopIteration
    # etc.
```

The `is` operator is faster than `==`, because it cannot be overloaded, so Python does not have to find and invoke special methods to evaluate it, and computing is as simple as comparing two integer IDs. In contrast, `a == b` is syntactic sugar for `a.__eq__(b)`. The `__eq__` method inherited from `object` compares object IDs, so it produces the same result as `is`. But most built-in types override `__eq__` with more meaningful implementations that actually take into account the values of the object attributes. Equality may involve a lot of processing—for example, when comparing large collections or deeply nested structures.

## WARNING

Usually we are more interested in object equality than identity. The **only** common proper case of `is` is checking for `None`. Most other uses I see while reviewing code are wrong. If you are not sure, use `==`. It's usually what you want, and also works with `None`—albeit not as fast.

To wrap up this discussion of identity versus equality, we'll see that the famously immutable `tuple` is not as rigid as you may expect.

## The Relative Immutability of Tuples

Tuples, like most Python collections—lists, dicts, sets, etc.—are containers: they hold references to objects.<sup>1</sup> If the referenced items are mutable, they may change even if the tuple itself does not. In other words, the immutability of tuples really refers to the physical contents of the `tuple` data structure (i.e., the references it holds), and does not extend to the referenced objects.

Example 6-5 illustrates the situation in which the value of a tuple changes as result of changes to a mutable object referenced in it. What can never change in a tuple is the identity of the items it contains.

*Example 6-5. `t1` and `t2` initially compare equal, but changing a mutable item inside tuple `t1` makes it different*

---

```
>>> t1 = (1, 2, [30, 40]) ❶
>>> t2 = (1, 2, [30, 40]) ❷
>>> t1 == t2 ❸
True
>>> id(t1[-1]) ❹
4302515784
>>> t1[-1].append(99) ❺
>>> t1
```

```
(1, 2, [30, 40, 99])
>>> id(t1[-1]) ❸
4302515784
>>> t1 == t2 ❹
False
```

- ❶ t1 is immutable, but t1[-1] is mutable.
- ❷ Build a tuple t2 whose items are equal to those of t1.
- ❸ Although distinct objects, t1 and t2 compare equal, as expected.
- ❹ Inspect the identity of the list at t1[-1].
- ❺ Modify the t1[-1] list in place.
- ❻ The identity of t1[-1] has not changed, only its value.
- ❼ t1 and t2 are now different.

This relative immutability of tuples is behind the riddle “[A += Assignment Puzzler](#)”. It’s also the reason why some tuples are unhashable, as we’ve seen in “[What Is Hashable?](#)”.

The distinction between equality and identity has further implications when you need to copy an object. A copy is an equal object with a different ID. But if an object contains other objects, should the copy also duplicate the inner objects, or is it OK to share them? There’s no single answer. Read on for a discussion.

## Copies Are Shallow by Default

The easiest way to copy a list (or most built-in mutable collections) is to use the built-in constructor for the type itself. For example:

```
>>> l1 = [3, [55, 44], (7, 8, 9)]
>>> l2 = list(l1) ❶
>>> l2
[3, [55, 44], (7, 8, 9)]
>>> l2 == l1 ❷
True
```

```
>>> l2 is l1 ❸  
False
```

- ❶ `list(l1)` creates a copy of `l1`.
- ❷ The copies are equal.
- ❸ But refer to two different objects.

For lists and other mutable sequences, the shortcut `l2 = l1[:]` also makes a copy.

However, using the constructor or `[:]` produces a *shallow copy* (i.e., the outermost container is duplicated, but the copy is filled with references to the same items held by the original container). This saves memory and causes no problems if all the items are immutable. But if there are mutable items, this may lead to unpleasant surprises.

In [Example 6-6](#), we create a shallow copy of a list containing another list and a tuple, and then make changes to see how they affect the referenced objects.

### TIP

If you have a connected computer on hand, I highly recommend watching the interactive animation for [Example 6-6](#) at the [Online Python Tutor](#). As I write this, direct linking to a prepared example at [pythontutor.com](#) is not working reliably, but the tool is awesome, so taking the time to copy and paste the code is worthwhile.

*Example 6-6. Making a shallow copy of a list containing another list; copy and paste this code to see it animated at the Online Python Tutor*

```
l1 = [3, [66, 55, 44], (7, 8, 9)]  
l2 = list(l1) ❶  
l1.append(100) ❷
```

```

l1[1].remove(55)    ❸
print('l1:', l1)
print('l2:', l2)
l2[1] += [33, 22]   ❹
l2[2] += (10, 11)   ❺
print('l1:', l1)
print('l2:', l2)

```

- ❶ l2 is a shallow copy of l1. This state is depicted in [Figure 6-3](#).
- ❷ Appending 100 to l1 has no effect on l2.
- ❸ Here we remove 55 from the inner list l1[1]. This affects l2 because l2[1] is bound to the same list as l1[1].
- ❹ For a mutable object like the list referred by l2[1], the operator += changes the list in place. This change is visible at l1[1], which is an alias for l2[1].
- ❺ += on a tuple creates a new tuple and rebinds the variable l2[2] here. This is the same as doing l2[2] = l2[2] + (10, 11). Now the tuples in the last position of l1 and l2 are no longer the same object. See [Figure 6-4](#).

The output of [Example 6-6](#) is [Example 6-7](#), and the final state of the objects is depicted in [Figure 6-4](#).

#### *Example 6-7. Output of Example 6-6*

```

l1: [3, [66, 44], (7, 8, 9), 100]
l2: [3, [66, 44], (7, 8, 9)]
l1: [3, [66, 44, 33, 22], (7, 8, 9), 100]
l2: [3, [66, 44, 33, 22], (7, 8, 9, 10, 11)]

```



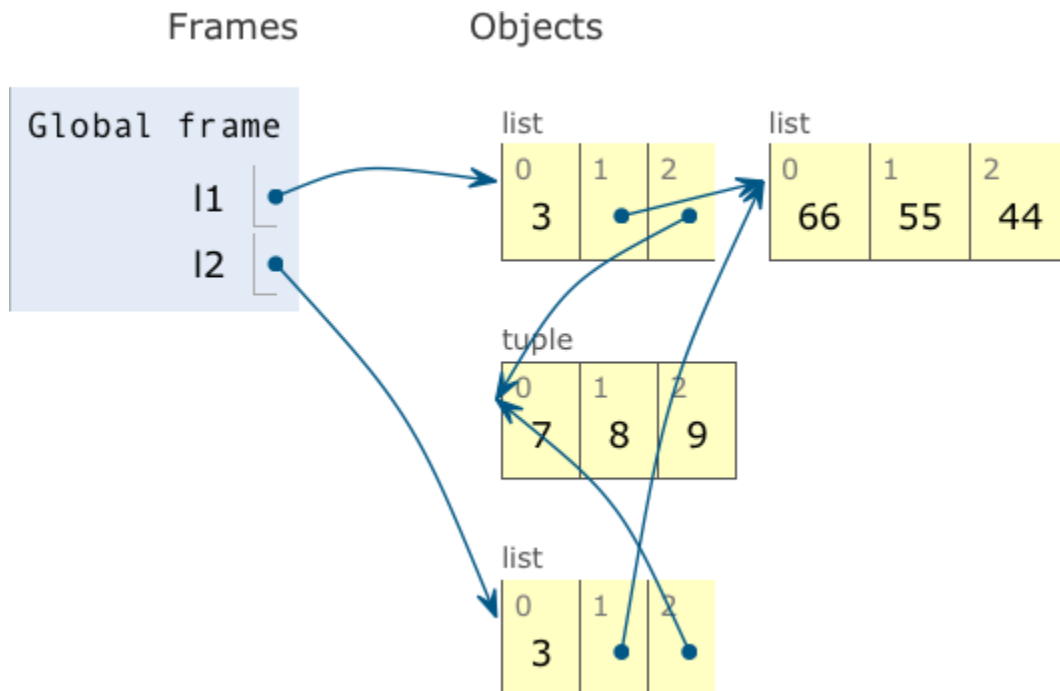


Figure 6-3. Program state immediately after the assignment `l2 = list(l1)` in Example 6-6. `l1` and `l2` refer to distinct lists, but the lists share references to the same inner list object [66, 55, 44] and tuple (7, 8, 9). (Diagram generated by the Online Python Tutor.)

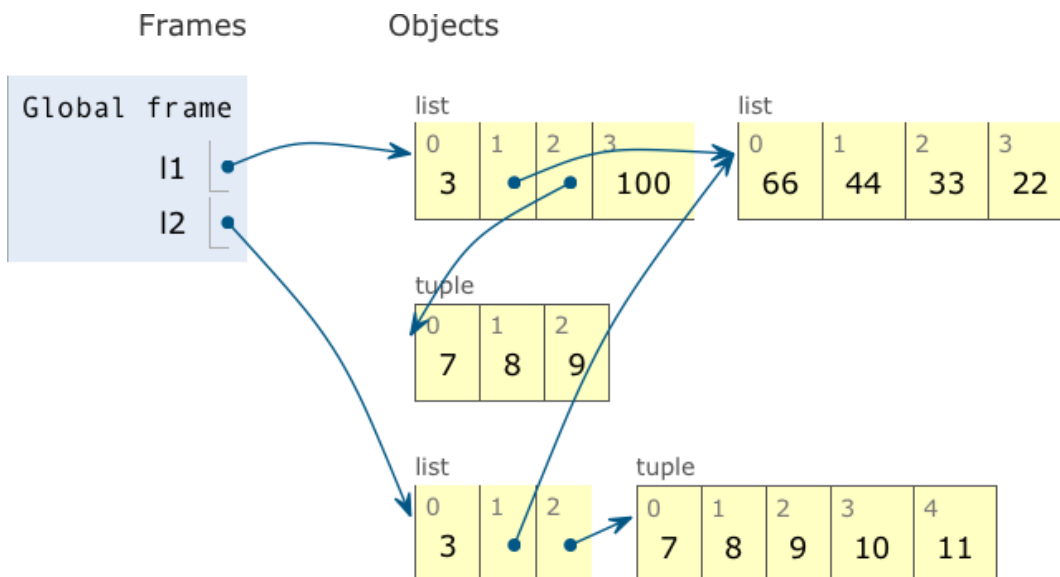


Figure 6-4. Final state of `l1` and `l2`: they still share references to the same list object, now containing [66, 44, 33, 22], but the operation `l2[2] += (10, 11)` created a new tuple with content (7, 8, 9, 10, 11), unrelated to the tuple (7, 8, 9) referenced by `l1[2]`. (Diagram generated by the Online Python Tutor.)

It should be clear now that shallow copies are easy to make, but they may or may not be what you want. How to make deep copies is our next topic.

## Deep and Shallow Copies of Arbitrary Objects

Working with shallow copies is not always a problem, but sometimes you need to make deep copies (i.e., duplicates that do not share references of embedded objects). The `copy` module provides the `deepcopy` and `copy` functions that return deep and shallow copies of arbitrary objects.

To illustrate the use of `copy()` and `deepcopy()`, [Example 6-8](#) defines a simple class, `Bus`, representing a school bus that is loaded with passengers and then picks up or drops off passengers on its route.

*Example 6-8. Bus picks up and drops off passengers*

---

**class** `Bus`:

```
def __init__(self, passengers=None):
    if passengers is None:
        self.passengers = []
    else:
        self.passengers = list(passengers)

def pick(self, name):
    self.passengers.append(name)

def drop(self, name):
    self.passengers.remove(name)
```

Now in the interactive [Example 6-9](#) we will create a `bus` object (`bus1`) and two clones—a shallow copy (`bus2`) and a deep copy (`bus3`)—to observe what happens as `bus1` drops off a student.

*Example 6-9. Effects of using `copy` versus `deepcopy`*

---

```

>>> import copy
>>> bus1 = Bus(['Alice', 'Bill', 'Claire', 'David'])
>>> bus2 = copy.copy(bus1)
>>> bus3 = copy.deepcopy(bus1)
>>> id(bus1), id(bus2), id(bus3)
(4301498296, 4301499416, 4301499752) ❶
>>> bus1.drop('Bill')
>>> bus2.passengers
['Alice', 'Claire', 'David'] ❷
>>> id(bus1.passengers), id(bus2.passengers),
id(bus3.passengers)
(4302658568, 4302658568, 4302657800) ❸
>>> bus3.passengers
['Alice', 'Bill', 'Claire', 'David'] ❹

```

- ❶ Using `copy` and `deepcopy`, we create three distinct `Bus` instances.
- ❷ After `bus1` drops 'Bill', he is also missing from `bus2`.
- ❸ Inspection of the `passengers` attributes shows that `bus1` and `bus2` share the same list object, because `bus2` is a shallow copy of `bus1`.
- ❹ `bus3` is a deep copy of `bus1`, so its `passengers` attribute refers to another list.

Note that making deep copies is not a simple matter in the general case. Objects may have cyclic references that would cause a naïve algorithm to enter an infinite loop. The `deepcopy` function remembers the objects already copied to handle cyclic references gracefully. This is demonstrated in [Example 6-10](#).

*Example 6-10. Cyclic references: `b` refers to `a`, and then is appended to `a`; `deepcopy` still manages to copy `a`*

---

```

>>> a = [10, 20]
>>> b = [a, 30]
>>> a.append(b)
>>> a
[10, 20, [[...], 30]]
>>> from copy import deepcopy
>>> c = deepcopy(a)

```

```
>>> c
[10, 20, [...], 30]]
```

Also, a deep copy may be too deep in some cases. For example, objects may refer to external resources or singletons that should not be copied. You can control the behavior of both `copy` and `deepcopy` by implementing the `__copy__()` and `__deepcopy__()` special methods as described in the [copy module documentation](#).

The sharing of objects through aliases also explains how parameter passing works in Python, and the problem of using mutable types as parameter defaults. These issues will be covered next.

## Function Parameters as References

The only mode of parameter passing in Python is *call by sharing*. That is the same mode used in most OO languages, including Ruby, Smalltalk, and Java (this applies to Java reference types; primitive types use call by value). Call by sharing means that each formal parameter of the function gets a copy of each reference in the arguments. In other words, the parameters inside the function become aliases of the actual arguments.

The result of this scheme is that a function may change any mutable object passed as a parameter, but it cannot change the identity of those objects (i.e., it cannot altogether replace an object with another). [Example 6-11](#) shows a simple function using `+=` on one of its parameters. As we pass numbers, lists, and tuples to the function, the actual arguments passed are affected in different ways.

---

*Example 6-11. A function may change any mutable object it receives*

```
>>> def f(a, b):
...     a += b
...     return a
... 
```

```

>>> x = 1
>>> y = 2
>>> f(x, y)
3
>>> x, y ❶
(1, 2)
>>> a = [1, 2]
>>> b = [3, 4]
>>> f(a, b)
[1, 2, 3, 4]
>>> a, b ❷
([1, 2, 3, 4], [3, 4])
>>> t = (10, 20)
>>> u = (30, 40)
>>> f(t, u) ❸
(10, 20, 30, 40)
>>> t, u
((10, 20), (30, 40))

```

- ❶ The number `x` is unchanged.
- ❷ The list `a` is changed.
- ❸ The tuple `t` is unchanged.

Another issue related to function parameters is the use of mutable values for defaults, as discussed next.

## Mutable Types as Parameter Defaults: Bad Idea

Optional parameters with default values are a great feature of Python function definitions, allowing our APIs to evolve while remaining backward-compatible. However, you should avoid mutable objects as default values for parameters.

To illustrate this point, in [Example 6-12](#), we take the `Bus` class from [Example 6-8](#) and change its `__init__` method to create `HauntedBus`. Here we tried to be clever and instead of having a default value of `passengers=None`, we have `passengers=[]`, thus avoiding the `if` in the previous `__init__`. This “cleverness” gets us into trouble.

*Example 6-12. A simple class to illustrate the danger of a mutable default*

---

```
class HauntedBus:
    """A bus model haunted by ghost passengers"""

    def __init__(self, passengers=[]): ❶
        self.passengers = passengers ❷

    def pick(self, name):
        self.passengers.append(name) ❸

    def drop(self, name):
        self.passengers.remove(name)
```

- ❶ When the `passengers` argument is not passed, this parameter is bound to the default list object, which is initially empty.
- ❷ This assignment makes `self.passengers` an alias for `passengers`, which is itself an alias for the default list, when no `passengers` argument is given.
- ❸ When the methods `.remove()` and `.append()` are used with `self.passengers` we are actually mutating the default list, which is an attribute of the function object.

Example 6-13 shows the eerie behavior of the `HauntedBus`.

*Example 6-13. Buses haunted by ghost passengers*

---

```
>>> bus1 = HauntedBus(['Alice', 'Bill'])
>>> bus1.passengers
['Alice', 'Bill']
>>> bus1.pick('Charlie')
>>> bus1.drop('Alice')
>>> bus1.passengers ❶
['Bill', 'Charlie']
>>> bus2 = HauntedBus() ❷
>>> bus2.pick('Carrie')
>>> bus2.passengers
['Carrie']
>>> bus3 = HauntedBus() ❸
```

```
>>> bus3.passengers ④
['Carrie']
>>> bus3.pick('Dave')
>>> bus2.passengers ⑤
['Carrie', 'Dave']
>>> bus2.passengers is bus3.passengers ⑥
True
>>> bus1.passengers ⑦
['Bill', 'Charlie']
```

- ① So far, so good: no surprises with `bus1`.
- ② `bus2` starts empty, so the default empty list is assigned to `self.passengers`.
- ③ `bus3` also starts empty, again the default list is assigned.
- ④ The default is no longer empty!
- ⑤ Now Dave, picked by `bus3`, appears in `bus2`.
- ⑥ The problem: `bus2.passengers` and `bus3.passengers` refer to the same list.
- ⑦ But `bus1.passengers` is a distinct list.

The problem is that `HauntedBus` instances that don't get an initial passenger list end up sharing the same passenger list among themselves.

Such bugs may be subtle. As [Example 6-13](#) demonstrates, when a `HauntedBus` is instantiated with passengers, it works as expected. Strange things happen only when a `HauntedBus` starts empty, because then `self.passengers` becomes an alias for the default value of the `passengers` parameter. The problem is that each default value is evaluated when the function is defined—i.e., usually when the module is loaded—and the default values become attributes of the function object. So if a default value is a mutable object, and you change it, the change will affect every future call of the function.

After running the lines in [Example 6-13](#), you can inspect the `HauntedBus.__init__` object and see the ghost students haunting its

`__defaults__` attribute:

```
>>> dir(HauntedBus.__init__) # doctest: +ELLIPSIS
['__annotations__', '__call__', ..., '__defaults__', ...]
>>> HauntedBus.__init__.__defaults__
(['Carrie', 'Dave'],)
```

Finally, we can verify that `bus2.passengers` is an alias bound to the first element of the `HauntedBus.__init__.__defaults__` attribute:

```
>>> HauntedBus.__init__.__defaults__[0] is bus2.passengers
True
```

The issue with mutable defaults explains why `None` is often used as the default value for parameters that may receive mutable values. In [Example 6-8](#), `__init__` checks whether the `passengers` argument is `None`, and assigns a new empty list to `self.passengers`. As explained in the following section, if `passengers` is not `None`, the correct implementation assigns a copy of it to `self.passengers`. Let's now take a closer look.

## Defensive Programming with Mutable Parameters

When you are coding a function that receives a mutable parameter, you should carefully consider whether the caller expects the argument passed to be changed.

For example, if your function receives a `dict` and needs to modify it while processing it, should this side effect be visible outside of the function or not? Actually it depends on the context. It's really a matter of aligning the expectation of the coder of the function and that of the caller.

The last bus example in this chapter shows how a `TwilightBus` breaks expectations by sharing its passenger list with its clients.



Before studying the implementation, see in [Example 6-14](#) how the `TwilightBus` class works from the perspective of a client of the class.

*Example 6-14. Passengers disappear when dropped by a `TwilightBus`*

---

```
>>> basketball_team = ['Sue', 'Tina', 'Maya', 'Diana', 'Pat']  
❶  
>>> bus = TwilightBus(basketball_team) ❷  
>>> bus.drop('Tina') ❸  
>>> bus.drop('Pat')  
>>> basketball_team ❹  
['Sue', 'Maya', 'Diana']
```

- ❶ `basketball_team` holds five student names.
- ❷ A `TwilightBus` is loaded with the team.
- ❸ The bus drops one student, then another.
- ❹ The dropped passengers vanished from the basketball team!

`TwilightBus` violates the “Principle of least astonishment,” a best practice of interface design. It surely is astonishing that when the bus drops a student, her name is removed from the basketball team roster.

[Example 6-15](#) is the implementation `TwilightBus` and an explanation of the problem.

*Example 6-15. A simple class to show the perils of mutating received arguments*

---

```
class TwilightBus:  
    """A bus model that makes passengers vanish"""  
  
    def __init__(self, passengers=None):  
        if passengers is None:  
            self.passengers = [] ❶  
        else:  
            self.passengers = passengers ❷  
  
    def pick(self, name):
```

```
self.passengers.append(name)
```

```
def drop(self, name):  
    self.passengers.remove(name) ❸
```

- ❶ Here we are careful to create a new empty list when `passengers` is `None`.
- ❷ However, this assignment makes `self.passengers` an alias for `passengers`, which is itself an alias for the actual argument passed to `__init__` (i.e., `basketball_team` in [Example 6-14](#)).
- ❸ When the methods `.remove()` and `.append()` are used with `self.passengers`, we are actually mutating the original list received as argument to the constructor.

The problem here is that the bus is aliasing the list that is passed to the constructor. Instead, it should keep its own passenger list. The fix is simple: in `__init__`, when the `passengers` parameter is provided, `self.passengers` should be initialized with a copy of it, as we did correctly in [Example 6-8](#) (“Deep and Shallow Copies of Arbitrary Objects”):

```
def __init__(self, passengers=None):  
    if passengers is None:  
        self.passengers = []  
    else:  
        self.passengers = list(passengers) ❶
```

- ❶ Make a copy of the `passengers` list, or convert it to a `list` if it’s not one.

Now our internal handling of the passenger list will not affect the argument used to initialize the bus. As a bonus, this solution is more flexible: now the argument passed to the `passengers` parameter may be a `tuple` or any other iterable, like a `set` or even database results, because the `list` constructor accepts any iterable. As we create our own list to manage, we ensure that it supports the necessary

`.remove()` and `.append()` operations we use in the `.pick()` and `.drop()` methods.

### TIP

Unless a method is explicitly intended to mutate an object received as argument, you should think twice before aliasing the argument object by simply assigning it to an instance variable in your class. If in doubt, make a copy. Your clients will often be happier.

## del and Garbage Collection

*Objects are never explicitly destroyed; however, when they become unreachable they may be garbage-collected.*

—Data Model, chapter of *The Python Language Reference*

The `del` statement deletes names, not objects. An object may be garbage collected as result of a `del` command, but only if the variable deleted holds the last reference to the object, or if the object becomes unreachable.<sup>2</sup> Rebinding a variable may also cause the number of references to an object to reach zero, causing its destruction.

## WARNING

There is a `__del__` special method, but it does not cause the disposal of the instance, and should not be called by your code. `__del__` is invoked by the Python interpreter when the instance is about to be destroyed to give it a chance to release external resources. You will seldom need to implement `__del__` in your own code, yet some Python beginners spend time coding it for no good reason. The proper use of `__del__` is rather tricky. See the `__del__` [special method documentation](#) in the “Data Model” chapter of *The Python Language Reference*.

In CPython, the primary algorithm for garbage collection is reference counting. Essentially, each object keeps count of how many references point to it. As soon as that *refcount* reaches zero, the object is immediately destroyed: CPython calls the `__del__` method on the object (if defined) and then frees the memory allocated to the object. In CPython 2.0, a generational garbage collection algorithm was added to detect groups of objects involved in reference cycles—which may be unreachable even with outstanding references to them, when all the mutual references are contained within the group. Other implementations of Python have more sophisticated garbage collectors that do not rely on reference counting, which means the `__del__` method may not be called immediately when there are no more references to the object. See “[PyPy, Garbage Collection, and a Deadlock](#)” by A. Jesse Jiryu Davis for discussion of improper and proper use of `__del__`.

To demonstrate the end of an object’s life, [Example 6-16](#) uses `weakref.finalize` to register a callback function to be called when an object is destroyed.

*Example 6-16. Watching the end of an object when no more references point to it*

---

```

>>> import weakref
>>> s1 = {1, 2, 3}
>>> s2 = s1 ❶
>>> def bye(): ❷
...     print('Gone with the wind...')
...
>>> ender = weakref.finalize(s1, bye) ❸
>>> ender.alive ❹
True
>>> del s1
>>> ender.alive ❺
True
>>> s2 = 'spam' ❻
Gone with the wind...
>>> ender.alive
False

```

- ❶ `s1` and `s2` are aliases referring to the same set, `{1, 2, 3}`.
- ❷ This function must not be a bound method of the object about to be destroyed or otherwise hold a reference to it.
- ❸ Register the `bye` callback on the object referred by `s1`.
- ❹ The `.alive` attribute is `True` before the `finalize` object is called.
- ❺ As discussed, `del` does not delete an object, just a reference to it.
- ❻ Rebinding the last reference, `s2`, makes `{1, 2, 3}` unreachable. It is destroyed, the `bye` callback is invoked, and `ender.alive` becomes `False`.

The point of [Example 6-16](#) is to make explicit that `del` does not delete objects, but objects may be deleted as a consequence of being unreachable after `del` is used.

You may be wondering why the `{1, 2, 3}` object was destroyed in [Example 6-16](#). After all, the `s1` reference was passed to the `finalize` function, which must have held on to it in order to monitor the object and invoke the callback. This works because `finalize` holds a *weak reference* to `{1, 2, 3}`, as explained in the next section.

## Weak References

The presence of references is what keeps an object alive in memory. When the reference count of an object reaches zero, the garbage collector disposes of it. But sometimes it is useful to have a reference to an object that does not keep it around longer than necessary. A common use case is a cache.

Weak references to an object do not increase its reference count. The object that is the target of a reference is called the *referent*. Therefore, we say that a weak reference does not prevent the referent from being garbage collected.

Weak references are useful in caching applications because you don't want the cached objects to be kept alive just because they are referenced by the cache.

Example 6-17 shows how a `weakref.ref` instance can be called to reach its referent. If the object is alive, calling the weak reference returns it, otherwise `None` is returned.

### TIP

Example 6-17 is a console session, and the Python console automatically binds the `_` variable to the result of expressions that are not `None`. This interfered with my intended demonstration but also highlights a practical matter: when trying to micro-manage memory we are often surprised by hidden, implicit assignments that create new references to our objects. The `_` console variable is one example. Traceback objects are another common source of unexpected references.

*Example 6-17. A weak reference is a callable that returns the referenced object or `None` if the referent is no more*

---

```

>>> import weakref
>>> a_set = {0, 1}
>>> wref = weakref.ref(a_set) ❶
>>> wref
<weakref at 0x100637598; to 'set' at 0x100636748>
>>> wref() ❷
{0, 1}
>>> a_set = {2, 3, 4} ❸
>>> wref() ❹
{0, 1}
>>> wref() is None ❺
False
>>> wref() is None ❻
True

```

- ❶ The `wref` weak reference object is created and inspected in the next line.
- ❷ Invoking `wref()` returns the referenced object, `{0, 1}`. Because this is a console session, the result `{0, 1}` is bound to the `_` variable.
- ❸ `a_set` no longer refers to the `{0, 1}` set, so its reference count is decreased. But the `_` variable still refers to it.
- ❹ Calling `wref()` still returns `{0, 1}`.
- ❺ When this expression is evaluated, `{0, 1}` lives, therefore `wref()` is not `None`. But `_` is then bound to the resulting value, `False`. Now there are no more strong references to `{0, 1}`.
- ❻ Because the `{0, 1}` object is now gone, this last call to `wref()` returns `None`.

The [weakref module documentation](#) makes the point that the `weakref.ref` class is actually a low-level interface intended for advanced uses, and that most programs are better served by the use of the `weakref` collections and `finalize`. In other words, consider using `WeakKeyDictionary`, `WeakValueDictionary`, `WeakSet`, and `finalize` (which use weak references internally) instead of creating and handling your own `weakref.ref` instances by hand. We just did that in [Example 6-17](#) in the hope that showing a single `weakref.ref`

in action could take away some of the mystery around them. But in practice, most of the time Python programs use the `weakref` collections.

The next subsection briefly discusses the `weakref` collections.

## The WeakValueDictionary Skit

The class `WeakValueDictionary` implements a mutable mapping where the values are weak references to objects. When a referred object is garbage collected elsewhere in the program, the corresponding key is automatically removed from `WeakValueDictionary`. This is commonly used for caching.

Our demonstration of a `WeakValueDictionary` is inspired by the classic *Cheese Shop* skit by Monty Python, in which a customer asks for more than 40 kinds of cheese, including cheddar and mozzarella, but none are in stock.<sup>3</sup>

Example 6-18 implements a trivial class to represent each kind of cheese.

*Example 6-18. Cheese has a kind attribute and a standard representation*

---

```
class Cheese:

    def __init__(self, kind):
        self.kind = kind

    def __repr__(self):
        return 'Cheese(%r)' % self.kind
```

In Example 6-19, each cheese is loaded from a `catalog` to a `stock` implemented as a `WeakValueDictionary`. However, all but one disappear from the `stock` as soon as the `catalog` is deleted. Can you



explain why the Parmesan cheese lasts longer than the others?<sup>4</sup> The tip after the code has the answer.

*Example 6-19. Customer: “Have you in fact got any cheese here at all?”*

```
>>> import weakref
>>> stock = weakref.WeakValueDictionary() ❶
>>> catalog = [Cheese('Red Leicester'), Cheese('Tilsit'),
...             Cheese('Brie'), Cheese('Parmesan')]
...
>>> for cheese in catalog:
...     stock[cheese.kind] = cheese ❷
...
>>> sorted(stock.keys())
['Brie', 'Parmesan', 'Red Leicester', 'Tilsit'] ❸
>>> del catalog
>>> sorted(stock.keys())
['Parmesan'] ❹
>>> del cheese
>>> sorted(stock.keys())
[]
```

- ❶ stock is a WeakValueDictionary.
- ❷ The stock maps the name of the cheese to a weak reference to the cheese instance in the catalog.
- ❸ The stock is complete.
- ❹ After the catalog is deleted, most cheeses are gone from the stock, as expected in WeakValueDictionary. Why not all, in this case?

## TIP

A temporary variable may cause an object to last longer than expected by holding a reference to it. This is usually not a problem with local variables: they are destroyed when the function returns. But in [Example 6-19](#), the `for` loop variable `cheese` is a global variable and will never go away unless explicitly deleted.

A counterpart to the `WeakValueDictionary` is the `WeakKeyDictionary` in which the keys are weak references. The [`weakref.WeakKeyDictionary` documentation](#) hints on possible uses:

*[A `WeakKeyDictionary`] can be used to associate additional data with an object owned by other parts of an application without adding attributes to those objects. This can be especially useful with objects that override attribute accesses.*

The `weakref` module also provides a `WeakSet`, simply described in the docs as “Set class that keeps weak references to its elements. An element will be discarded when no strong reference to it exists any more.” If you need to build a class that is aware of every one of its instances, a good solution is to create a class attribute with a `WeakSet` to hold the references to the instances. Otherwise, if a regular `set` was used, the instances would never be garbage collected, because the class itself would have strong references to them, and classes live as long as the Python process unless you deliberately delete them.

These collections, and weak references in general, are limited in the kinds of objects they can handle. The next section explains.

## Limitations of Weak References

Not every Python object may be the target, or referent, of a weak reference. Basic `list` and `dict` instances may not be referents, but a

plain subclass of either can solve this problem easily:

```
class MyList(list):  
    """list subclass whose instances may be weakly  
    referenced"""  
  
a_list = MyList(range(10))  
  
# a_list can be the target of a weak reference  
wref_to_a_list = weakref.ref(a_list)
```

A `set` instance can be a referent, and that's why a `set` was used in [Example 6-17](#). User-defined types also pose no problem, which explains why the silly `Cheese` class was needed in [Example 6-19](#). But `int` and `tuple` instances cannot be targets of weak references, even if subclasses of those types are created.

Most of these limitations are implementation details of CPython that may not apply to other Python interpreters. They are the result of internal optimizations, some of which are discussed in the following (highly optional) section.

## Tricks Python Plays with Immutables

### NOTE

This optional section discusses some Python details that are not really important for *users* of Python, and that may not apply to other Python implementations or even future versions of CPython. Nevertheless, I've seen people stumble upon these corner cases and then start using the `is` operator incorrectly, so I felt they were worth mentioning.

I was surprised to learn that, for a tuple `t`, `t[:]` does not make a copy, but returns a reference to the same object. You also get a reference to the same tuple if you write `tuple(t)`.<sup>5</sup> [Example 6-20](#) proves it.

*Example 6-20. A tuple built from another is actually the same exact tuple*

---

```
>>> t1 = (1, 2, 3)
>>> t2 = tuple(t1)
>>> t2 is t1 ❶
True
>>> t3 = t1[:]
>>> t3 is t1 ❷
True
```

- ❶ `t1` and `t2` are bound to the same object.
- ❷ And so is `t3`.

The same behavior can be observed with instances of `str`, `bytes`, and `frozenset`. Note that a `frozenset` is not a sequence, so `fs[:]` does not work if `fs` is a `frozenset`. But `fs.copy()` has the same effect: it cheats and returns a reference to the same object, and not a copy at all, as [Example 6-21](#) shows.<sup>6</sup>

*Example 6-21. String literals may create shared objects*

---

```
>>> t1 = (1, 2, 3)
>>> t3 = (1, 2, 3) ❶
>>> t3 is t1 ❷
False
>>> s1 = 'ABC'
>>> s2 = 'ABC' ❸
>>> s2 is s1 ❹
True
```

- ❶ Creating a new tuple from scratch.
- ❷ `t1` and `t3` are equal, but not the same object.
- ❸ Creating a second `str` from scratch.

④ Surprise: `a` and `b` refer to the same `str`!

The sharing of string literals is an optimization technique called *interning*. CPython uses the same technique with small integers to avoid unnecessary duplication of numbers that appear frequently in programs like 0, 1, -1, etc. Note that CPython does not intern all strings or integers, and the criteria it uses to do so is an undocumented implementation detail.

**WARNING**

Never depend on `str` or `int` interning! Always use `==` and not `is` to compare them for equality. Interning is an optimization for internal use of the Python interpreter.

The tricks discussed in this section, including the behavior of `frozenset.copy()`, are harmless “lies”; they save memory and make the interpreter faster. Do not worry about them, they should not give you any trouble because they only apply to immutable types. Probably the best use of these bits of trivia is to win bets with fellow Pythonistas.

## Chapter Summary

Every Python object has an identity, a type, and a value. Only the value of an object changes over time.<sup>7</sup>

If two variables refer to immutable objects that have equal values (`a == b` is `True`), in practice it rarely matters if they refer to copies or are aliases referring to the same object because the value of an immutable object does not change, with one exception. The exception is immutable collections such as tuples and frozensets: if an immutable collection holds references to mutable items, then its value may actually change when the value of a mutable item changes. In practice, this scenario is not so common. What never changes in an immutable collection are the identities of the objects within.

The fact that variables hold references has many practical consequences in Python programming:

- Simple assignment does not create copies.
- Augmented assignment with `+=` or `*=` creates new objects if the lefthand variable is bound to an immutable object, but may modify a mutable object in place.
- Assigning a new value to an existing variable does not change the object previously bound to it. This is called a rebinding: the variable is now bound to a different object. If that variable was the last reference to the previous object, that object will be garbage collected.
- Function parameters are passed as aliases, which means the function may change any mutable object received as an argument. There is no way to prevent this, except making local copies or using immutable objects (e.g., passing a tuple instead of a list).

- Using mutable objects as default values for function parameters is dangerous because if the parameters are changed in place, then the default is changed, affecting every future call that relies on the default.

In CPython, objects are discarded as soon as the number of references to them reaches zero. They may also be discarded if they form groups with cyclic references but no outside references. In some situations, it may be useful to hold a reference to an object that will not—by itself—keep an object alive. One example is a class that wants to keep track of all its current instances. This can be done with weak references, a low-level mechanism underlying the more useful collections `WeakValueDictionary`, `WeakKeyDictionary`, `WeakSet`, and the `finalize` function from the `weakref` module.

## Further Reading

The “Data Model” chapter of *The Python Language Reference* starts with a clear explanation of object identities and values.

Wesley Chun, author of the *Core Python* series of books, made a great presentation about many of the topics covered in this chapter during OSCON 2013. You can download the slides from the [“Python 103: Memory Model & Best Practices” talk page](#). There is also a [YouTube video](#) of a longer presentation Wesley gave at EuroPython 2011, covering not only the theme of this chapter but also the use of special methods.

Doug Hellmann wrote a long series of excellent blog posts titled [Python Module of the Week](#), which became a book, *The Python Standard Library by Example*. His posts [“copy – Duplicate Objects”](#) and [“weakref – Garbage-Collectable References to Objects”](#) cover some of the topics we just discussed.

More information on the CPython generational garbage collector can be found in the [gc module documentation](#), which starts with the sentence “This module provides an interface to the optional garbage collector.” The “optional” qualifier here may be surprising, but the [“Data Model” chapter](#) also states:

*An implementation is allowed to postpone garbage collection or omit it altogether—it is a matter of implementation quality how garbage collection is implemented, as long as no objects are collected that are still reachable.*

Fredrik Lundh—creator of key libraries like ElementTree, Tkinter, and the PIL image library—has a short post about the Python garbage collector titled [“How Does Python Manage Memory?”](#) He emphasizes that the garbage collector is an implementation feature that behaves differently across Python interpreters. For example, Jython uses the Java garbage collector.

The CPython 3.4 garbage collector improved handling of objects with a `__del__` method, as described in [PEP 442 — Safe object finalization](#).

Wikipedia has an article about [string interning](#), mentioning the use of this technique in several languages, including Python.



## SOAPBOX

### Equal Treatment to All Objects

I learned Java before I discovered Python. The `==` operator in Java never felt right for me. It is much more common for programmers to care about equality than identity, but for objects (not primitive types) the Java `==` compares references, and not object values. Even for something as basic as comparing strings, Java forces you to use the `.equals` method. Even then, there is another catch: if you write `a.equals(b)` and `a` is `null`, you get a null pointer exception. The Java designers felt the need to overload `+` for strings, so why not go ahead and overload `==` as well?

Python gets this right. The `==` operator compares object values and `is` compares references. And because Python has operator overloading, `==` works sensibly with all objects in the standard library, including `None`, which is a proper object, unlike Java's `null`.

And of course, you can define `__eq__` in your own classes to decide what `==` means for your instances. If you don't override `__eq__`, the method inherited from `object` compares object IDs, so the fallback is that every instance of a user-defined class is considered different.

These are some of the things that made me switch from Java to Python as soon as I finished reading the Python Tutorial one afternoon in September 1998.

### Mutability

This chapter would be redundant if all Python objects were immutable. When you are dealing with unchanging objects, it makes no difference whether variables hold the actual objects or references to shared objects. If `a == b` is true, and neither object can change, they might as well be the same. That's why string interning is safe. Object identity becomes important only when objects are mutable.

In “pure” functional programming, all data is immutable: appending to a collection actually creates a new collection. Python, however, is not a functional language, much less a pure one. Instances of user-defined classes are mutable by default in Python—as in most object-oriented languages. When creating your own objects, you have to be extra careful to make them immutable, if that is a requirement. Every attribute of the object must also be immutable, otherwise you end up with something like the `tuple`: immutable as far as object IDs go, but the value of a `tuple` may change if it holds a mutable object.

Mutable objects are also the main reason why programming with threads is so hard to get right: threads mutating objects without proper synchronization produce corrupted data. Excessive synchronization, on the other hand, causes deadlocks.

## Object Destruction and Garbage Collection

There is no mechanism in Python to directly destroy an object, and this omission is actually a great feature: if you could destroy an object at any time, what would happen to existing strong references pointing to it?

Garbage collection in CPython is done primarily by reference counting, which is easy to implement, but is prone to memory leaking when there are reference cycles, so with version 2.0 (October 2000) a generational garbage collector was implemented, and it is able to dispose of unreachable objects kept alive by reference cycles.

But the reference counting is still there as a baseline, and it causes the immediate disposal of objects with zero references. This means that, in CPython—at least for now—it's safe to write this:

```
open('test.txt', 'wt', encoding='utf-8').write('1, 2, 3')
```

That code is safe because the reference count of the file object will be zero after the `write` method returns, and Python will immediately close the file before destroying the object representing it in memory. However, the same line is not safe in Jython or IronPython that use the garbage collector of their host runtimes (the Java VM and the .NET CLR), which are more sophisticated but do not rely on reference counting and may take longer to destroy the object and close the file. In all cases, including CPython, the best practice is to explicitly close the file, and the most reliable way of doing it is using the `with` statement, which guarantees that the file will be closed even if exceptions are raised while it is open. Using `with`, the previous snippet becomes:

```
with open('test.txt', 'wt', encoding='utf-8') as fp:  
    fp.write('1, 2, 3')
```

If you are into the subject of garbage collectors, you may want to read Thomas Perl's paper [“Python Garbage Collector Implementations: CPython, PyPy and GaS”](#), from which I learned the bit about the safety of the `open().write()` in CPython.

## Parameter Passing: Call by Sharing

A popular way of explaining how parameter passing works in Python is the phrase: “Parameters are passed by value, but the values are references.” This is not wrong, but causes confusion because the most common parameter passing modes in older languages are *call by value* (the function gets a copy of the argument) and *call by reference* (the function gets a pointer to the argument). In Python, the function gets a copy of the arguments, but the arguments are always references. So the value of the referenced objects may be changed, if they are mutable, but their identity cannot. Also, because the function gets a copy of the reference in an argument, rebinding it has no effect outside of the function. I adopted the term *call by sharing* after reading up on the subject in *Programming Language Pragmatics, Third Edition* by Michael L. Scott (Morgan Kaufmann), particularly “8.3.1: Parameter Modes.”

### The Full Quote of Alice and the Knights’s Song

I love this passage, but it was too long as a chapter opener. So here is the complete dialog about the Knight’s song, its name, and how the song and its name are called:

*‘You are sad,’ the Knight said in an anxious tone: ‘let me sing you a song to comfort you.’*

*‘Is it very long?’ Alice asked, for she had heard a good deal of poetry that day.*

*‘It’s long,’ said the Knight, ‘but very, VERY beautiful. Everybody that hears me sing it—either it brings the TEARS into their eyes, or else—’*

*‘Or else what?’ said Alice, for the Knight had made a sudden pause.*

*‘Or else it doesn’t, you know. The name of the song is called “HADDOCKS’ EYES”.’*

*‘Oh, that’s the name of the song, is it?’ Alice said, trying to feel interested.*

*‘No, you don’t understand,’ the Knight said, looking a little vexed. ‘That’s what the name is CALLED. The name really IS “THE AGED AGED MAN”.’*

*‘Then I ought to have said “That’s what the SONG is called”?’ Alice corrected herself.*

*‘No, you oughtn’t: that’s quite another thing! The SONG is called “WAYS AND MEANS”: but that’s only what it’s CALLED, you know!’*

*‘Well, what IS the song, then?’ said Alice, who was by this time completely bewildered.*

*‘I was coming to that,’ the Knight said. ‘The song really IS “A-SITTING ON A GATE”: and the tune’s my own invention.’*

—Lewis Carroll, *Through the Looking-Glass*, Chapter VIII,  
“It’s My Own Invention”

- 
- 1 On the other hand, flat sequences like `str`, `bytes`, and `array.array` don't contain references but physically hold their data—characters, bytes, and numbers—in contiguous memory.
  - 2 If two objects refer to each other, as in [Example 6-10](#), they may be destroyed if the garbage collector determines that they are otherwise unreachable because their only references are their mutual references.
  - 3 `cheeseshop.python.org` is also an alias for PyPI—the Python Package Index software repository—which started its life quite empty. At the time of this writing, the Python Cheese Shop has 41,426 packages. Not bad, but still far from the more than 131,000 modules available in CPAN—the Comprehensive Perl Archive Network—the envy of all dynamic language communities.
  - 4 Parmesan cheese is aged at least a year at the factory, so it is more durable than fresh cheese, but this is not the answer we are looking for.
  - 5 This is clearly documented. Type `help(tuple)` in the Python console to read: “If the argument is a tuple, the return value is the same object.” I thought I knew everything about tuples before writing this book.
  - 6 The harmless lie of having the `copy` method not copying anything can be justified by interface compatibility: it makes `frozenset` more compatible with `set`. Anyway, it makes no difference to the end user whether two identical immutable objects are the same or are copies.
  - 7 Actually the type of an object may be changed by merely assigning a different class to its `__class__` attribute, but that is pure evil and I regret writing this footnote.